AdaCore Technologies for

CENELEC EN 50128:2011

Jean-Louis Boulanger and
Quentin Ochem, for AdaCore

AdaCore
Build software that matters.

ADACORE TECHNOLOGIES FOR CENELEC EN 50128:2011

VERSION 1.0

JEAN-LOUIS BOULANGER AND QUENTIN OCHEM, FOR ADACORE

October 6, 2015

About the Authors

Jean-Louis Boulanger

For more than 15 years 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 50128:2011 and IEC 62279 standards. He has also served as a professor and researcher at the University of Technology of Compiègne.

Quentin Ochem

Quentin Ochem has a software engineering background, with a special focus on development and verification tools for safety- and mission-critical systems. He has over 10 years of experience in the Ada programming language. He has conducted customer training on topics including the Ada language, AdaCore tools, and the DO-178B and DO-178C software certification standards. He is currently leading the technical account management and business development activities at AdaCore, in connection with projects from the avionics, railroad, space, and defense industries, both in Europe and North America.

CONTENTS

CHAPTER 1

Foreword

This document presents the usage of AdaCore's technology in conjunction with the CENELEC EN 50128:2011 standard. It describes where the technology fits best and how it can best be used to meet various requirements of the standard.

AdaCore's technology revolves around programming activities, as well as the closely-related design and verification activities. This is the bottom of the V cycle as defined by section 5.3 in EN 50128. It is based on the features of the Ada language (highly recommended by table A.15), in particular its 2012 revision, which adds some significant capabilities in terms of specification and verification.

AdaCore's technology brings two main benefits to a CENELEC EN 50128 process: first, the ability to write software interface specifications and software component specifications directly in the source code. Interfaces can be formally expressed in such forms as strong typing, parameter constraints, and subprogram contracts. This specification can be used to clarify interface documentation, enforce certain constraints while programming, and to provide an extensive foundation for software component and integration verification.

The other benefit targets the verification activities. Bringing additional specification at the language level allows verification activities to run earlier in the process, 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 technology can be used at all levels, from SIL0 to SIL4. At lower levels, the full Ada language is suitable, independent of platform. At higher levels, specific subsets will be needed, in particular the Ravenscar [BUN 04; MCC 11] subset for concurrent semantics or the Zero-Footprint profile [GNA 01] to reduce the language to a subset with no run-time library requirements. At the highest level, the SPARK language [MCC 15], along with the SPARK verification toolsuite, allows mathematical proof of properties ranging from absence of run-time exceptions to correctness of the implementation against a formally defined specification.

The following tools and technology will be presented:

- The Ada 2012 language, which is a compilable imperative language with strong specification and verification features. We'll refer to this version of the language simply as Ada;
- The SPARK 2014 language, a subset of Ada allowing formal verification. We'll refer to this version of the language simply as SPARK;
- The SPARK 2014 verification toolset, performing formal proof and verification on code written in SPARK. We'll refer to this version of the toolset simply as the SPARK toolset;
- The GNAT compiler, which compiles the Ada (and thus SPARK) languages;
- CodePeer - a static analysis tool that identifies potential run-time errors in Ada code;
- GNATmetric - a metric computation tool;
- GNATcheck - a coding standard checker;
- GNATdashboard - a metric integration and management platform;
- GNATtest - a unit testing framework generator;
- GNATemulator - a processor emulator;
- GNATcoverage - a structural code coverage checker;
- QGen - a Mathworks® Simulink® / Stateflow® code generator;

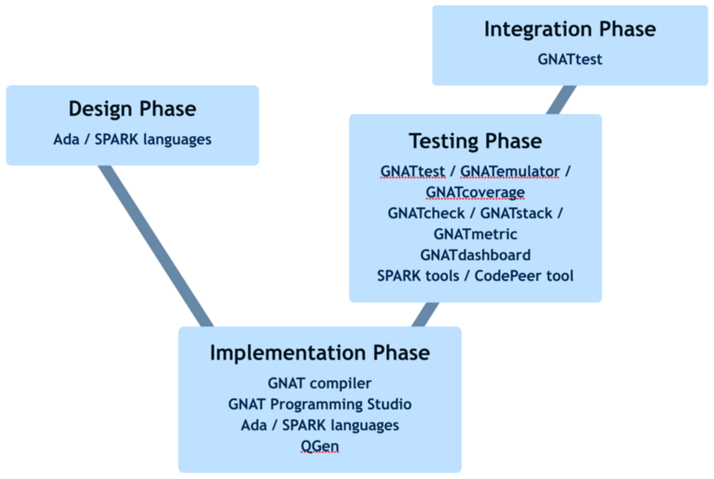


Figure 1.1: Contributions of AdaCore tools to the V cycle See [WWW 01] for a description of the high integrity product line for railway software

CHAPTER 2

CENELEC EN 50128

2.1 Introduction to the Standard

Today, railway projects are subject to a legal framework (laws, decrees, etc.) and also a normative process based on certification standards (CENELEC EN 50126 [CEN 00], EN 50129 [CEN 03] and EN 50128 [CEN 01; CEN 11]) that define certain objectives in terms of RAMS (Reliability, Availability, Maintainability and Safety).

The three standards are concerned with the safety-related aspects of the system, down to the hardware and/or software elements used. Figure 2.1 depicts the scope of the various CENELEC standards.

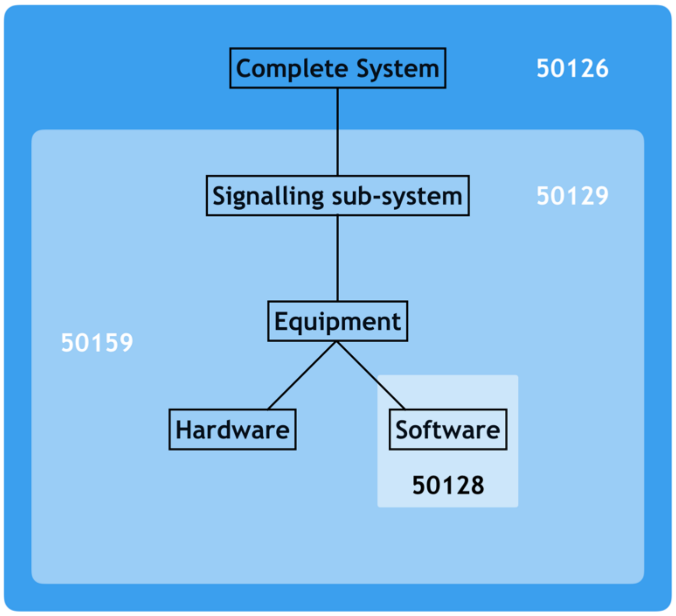


Figure 2.1: Main standards applicable to railway systems

CENELEC EN 50128 [CEN 11] specifies the procedures and prerequisites (organization, independence and competency management, quality management, V&V team, etc.) applicable to the development of programmable electronic systems used in railway control and protection applications. CENELEC EN 50128 therefore may apply to some software applications in the rail sector but not necessarily to all.

CENELEC EN 50128 is used in both safety-related and non-safety-related domains – for this reason, CENELEC EN 50128 introduces Software Safety Integrity Level SSIL 0, which pertains to non-safety-related software applications – and applies exclusively to software and the interaction of a software application with the whole system. This standard recommends the implementation of a V-lifecycle, from the software specification to the overall software testing. One of the distinctive points of CENELEC EN 50128 is its requirement to justify the implementation of the resources. For this reason, it is said to be a "resources standard".

CENELEC EN 50128 explicitly introduces the concept of assessment. As shown in [BOU 07], for software applications the assessment process involves demonstrating that the software application achieves its associated safety objectives.

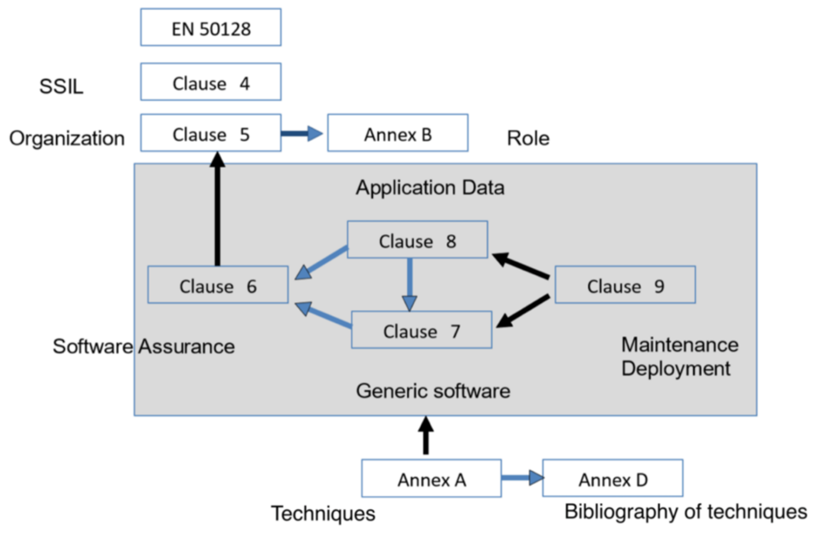


Figure 2.2: Structure of CENELEC EN 50128:2011

Figure 2.2 illustrates the structure of the 2011 version of CENELEC EN 50128. This standard introduces (clause 6) the concept of software assurance (SwA), whose goal is to achieve a software package with a minimum level of error. Software assurance involves Quality Assurance, skill evaluation, verification and validation, and independent assessment. CENELEC EN 50128 makes a clear separation between the application data (clause 8) and the software (clause 7), which is then called the generic software. One of the important points in the 2011 version of CENELEC EN 50128 is the addition of Clause 9, which is concerned with the software's maintenance and deployment. CENELEC 50128:2001 introduced a requirement that the compilers be purpose-certified, but did not give any clear indication of what precisely was expected. CENELEC 50128:2011 formally introduces the need to demonstrate the qualification for the tools employed for a project (see section 6.7 of the standard). Three classes of tools are introduced: T1, T2 and T3.

The T1 category is reserved for tools which affect neither the verification 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 from category T2 are tools used for verifying compliance with a coding standard, generating quantified metrics, performing static analysis of the source code, managing and executing tests, etc. The category T3 applies to tools which, if faulty, could have an impact on (and, for example, introduce errors into) the final executable software. This class includes compilers, code generators, etc.

Section 6.7 of CENELEC 50128:2011 defines a set of recommendations for each category; these affect the content of the tool qualification report.

Class of Tools	Applicable sections(s)	Step
T1	6.7.4.1	Identification
T2	6.7.4.1	Identification
	6.7.4.2	Justification
	6.7.4.3	Specification
	6.7.4.10, 6.7.4.11	Version management
T3	6.7.4.1	Identification
	6.7.4.2	Justification
	6.7.4.3	Specification
	(6.7.4.4 and 6.7.4.5) or 6.7.4.6	Conformity proof
	(6.7.4.7 or 6.7.4.8) and 6.7.4.9	Requirement fulfillment
	6.7.4.10, 6.7.4.11	Version management

Table 1: Tool qualification recommendations

CENELEC 50128:2011 identifies 12 requirements (numbered from 6.7.4.1 to 6.7.4.12) concerning tool qualification. Requirement 6.7.4.12 is linked to Table 1, which a correction of the published version. The steps shown in Table 1 indicate the requirements to be met and reflect the additional effort needed as the tool level increases (for more information see [BOU 15 – Chapter 9]).

2.2 AdaCore Qualification Methodology

Tool qualification is a requirement of CENELEC EN 50128:2011 (6.7.x). Verification tools are required to be qualified at the T2 level. Code generators and compilers need to be qualified at the T3 level.

Similar qualification can be found in avionics: TQL-5 or TQL-4 for verification tools and TQL-4 to TQL-1 for code generators.

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

In addition to the above, tools can be provided in a special subscription, called “sustained”. In this mode, a specific version of the tools can be put into special maintenance, where AdaCore retains the ability to investigate known problems and fix potential issues on these branches for several years.

CHAPTER 3

TOOLS AND TECHNOLOGIES OVERVIEW

3.1 Ada

Ada is a modern programming language designed for large, long-lived applications – and embedded systems in particular – where reliability 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 Dr. 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 Mr. 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 most recent version of the language standard, Ada 2012 [ADA 12; BAR 13; BAR 14].

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

Ada is seeing significant usage worldwide in high-integrity / safety-critical / high-security domains including commercial and military aircraft avionics, air traffic control, railroad systems, 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.

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

3.1.1 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:

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, or 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.

Programming in the large

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

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.

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 OO 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 provided additional OOP features including Java-like interfaces and traditional 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 [GNA 13] for more details.

Concurrent programming

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

Systems programming

Both in the "core" language and the Systems Programming Annex, Ada supplies the necessary features to allow the programmer to get close to the hardware. For example, you can specify the bit layout for fields in a record, define the alignment and size, place data at specific machine addresses, and express specialized or time-critical code sequences in assembly language. You can also write interrupt handlers in Ada, using the protected type facility.

Real-time programming

Ada's tasking features allow you to express common real-time idioms (periodic tasks, event-driven tasks), and the Real-Time Annex provides several facilities that allow you to 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, and Ada 2005 introduced several others including Earliest Deadline First.

High-integrity systems

With its emphasis on sound software engineering principles Ada supports the development of high-integrity applications, including those that need to be certified against safety standards such as EN 50128 for rail systems and DO-178B and DO-178C for avionics, and security standards such as the Common Criteria. For example, strong typing means that data intended for one purpose will not be accessed via inappropriate operations; errors such as treating pointers as integers (or vice versa) are prevented. And Ada's array bounds checking prevents buffer overrun vulnerabilities that are common in C and C++.

However, the full language is inappropriate in a safety-critical application, since the generality and flexibility may interfere with traceability / certification requirements. Ada addresses this issue by supplying a compiler directive, pragma Restrictions, that allows you to constrain 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 practical. Ada 2012 has 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.

3.1.2 Ada Benefits Summary

- Helps you design safe and reliable code
- Reduces development costs
- Supports new and changing technologies
- Facilitates development of complex programs
- Helps make code readable and portable
- Reduces certification costs for safety-critical software

3.1.3 Ada Features Summary

- Object-Oriented programming
- Strong typing
- Abstractions to fit program domain
- Generic programming/templates
- Exception handling
- Facilities for modular organization of code
- Standard libraries for I/O, string handling, numeric computing, containers
- Systems programming
- Concurrent programming
- Real-time programming
- Distributed systems programming
- Numeric processing
- Interfaces to other languages (C, COBOL, Fortran)

In brief, Ada is an internationally standardized language combining object-oriented programming features, well- engineered concurrency facilities, real-time support, and built-in reliability. As such it is an appropriate tool for addressing the real issues facing software developers today. Ada is used throughout a number of major industries to design software that protects businesses and lives.

3.2 SPARK

SPARK is a software development technology specifically designed for engineering high-reliability applications.

It consists of a programming language and a verification toolset designed for ultra-low defect software, for example where safety and security are key requirements.

SPARK has an impressive industrial track record. Since its inception in the late 1980s it has been applied worldwide in a range of industrial applications such as civil and military avionics, railway signaling, cryptographic and cross-domain solutions. SPARK 2014 is the most recent version of this leading software technology, explained in [MCC 15].

3.2.1 Flexibility

SPARK 2014 offers the flexibility of configuring the language on a per-project basis - applying restrictions that allow the fine-tuning of the permitted language features as appropriate to coding standards or run-time environments.

SPARK 2014 code can easily be combined with full Ada code or with C, meaning that new systems can be built on and reuse legacy code bases.

3.2.2 Powerful Static Verification

The SPARK 2014 language supports a wide range of different types of static verification. At one end of the spectrum is basic data and control flow analysis ie. exhaustive detection of uninitialized variables and ineffective assignment. For more critical applications, dependency contracts can be used to 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 is designed to support mathematical proof and thus offers access to a range of verification objectives: proving the absence of run-time exceptions, proving safety or security properties, or proving that the software implementation meets a formal specification of the program's required behavior.

3.2.3 Ease of Adoption

SPARK 2014 is an easy-to-adopt approach to increasing the reliability of your software. Software engineers will find the SPARK 2014 language contains the powerful programming language features with which they are familiar, making the language easy to learn.

SPARK 2014 converges its contract syntax for functional behavior with that of Ada 2012. Programmers familiar with writing executable contracts for run-time assertion checking will find the same paradigm can be applied for writing contracts that can be verified statically (ie. pre-compilation and pre-test) using automated tools.

3.2.4 Reduced Cost of Unit Testing

The costs associated with the demanding levels of unit testing required for high-assurance software - particularly in the context of industry standards such as CENELEC EN 50128 - are a major contribution to high delivery costs for safety-critical software. SPARK 2014 presents an innovative solution to this problem by allowing automated proof to be used in combination with unit testing to demonstrate functional correctness at subprogram level. In the high proportion of cases where proofs can be discharged automatically the cost of writing unit tests is completely avoided.

3.3 GNAT Pro Safety-Critical

GNAT Pro is a robust and flexible Ada development environment. It comprises a full Ada compiler (Ada 2012/2005/95/83 features) based on the GNU GCC technology, an Integrated Development Environment (GNAT Programming Studio and/or GNATbench), a comprehensive toolsuite including a visual debugger, and a set of libraries and bindings.

3.3.1 Configurable Run-Time Library

Using GNAT Pro Safety-Critical’s configurable run-time capability, you can specify any level of support for Ada’s dynamic features, from none at all to the full Ada 95, Ada 2005, or Ada 2012 language versions. The units included in the library may be either a subset of the standard units provided with GNAT Pro, or they may be specially tailored to the application. This capability is useful, for example, if one of the predefined profiles provides almost all the features needed to adapt an existing system to new safety-critical requirements, and where the costs of adaptation without the additional features are considered prohibitive.

3.3.2 Full Ada 2005 / 2012 Implementation

GNAT Pro provides a complete implementation of the Ada 2012 language. Developers of safety-critical and high- security systems can thus take advantage of features such as contract-based programming.

3.3.3 Simplification of Certification Effort

You can restrict language features that, although not requiring a run-time library, nevertheless could complicate the test coverage analysis part of the certification effort. For example, you can prohibit the use of constructs that would result in code with implicit loops and conditionals (such as a slice assignment).

3.3.4 Traceability

Through a compiler switch you can generate a low-level version of the source program that reveals implementation decisions but stays basically machine independent. This helps support traceability requirements, and may be used as a reference point for verifying that the object code matches the source code. Another compiler switch produces details of data representation (sizes, record layout, etc.), which is also helpful in traceability.

3.3.5 Safety-Critical Support and Expertise

At the heart of every AdaCore subscription are the support services we provide to our customers. AdaCore staff are recognized experts on the Ada language, certification standards, compilation technologies, and static and dynamic verification. They have extensive experience in supporting customers in avionics, railway, energy, space, air traffic management 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 advice 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 or ports to new platforms are also available.

3.4 CodePeer

CodePeer is an Ada source code analyzer that detects run-time and logic errors. It assesses potential bugs before program execution, serving as an automated peer reviewer, helping to find errors efficiently and early in the development life-cycle. It can also be used to perform impact analysis when introducing changes to the existing code, as well as helping vulnerability analysis. Using control-flow, data-flow, and other advanced static analysis techniques, CodePeer detects errors that would otherwise only be found through labor-intensive debugging.

3.4.1 Detects Errors before They Grow into Expensive Problems

CodePeer's advanced static error detection finds bugs in programs before programs are run. By mathematically analyzing every line of software, considering every possible input, and every path through the program, CodePeer can be used very early in the development life-cycle to identify problems when defects are much less costly to repair. It can also be used retrospectively on existing code bases, to detect latent vulnerabilities.

CodePeer is a standalone tool that may be used with any Ada compiler or fully integrated into the GNAT Pro development environment. It can detect several of the "Top 25 Most Dangerous Software Errors" in the Common Weakness Enumeration: CWE-120 (Classic Buffer Overflow), CWE-131 (Incorrect Calculation of Buffer Size), and CWE-190 (Integer Overflow or Wraparound). See [BLA 11] for more details.

3.4.2 Qualified for usage in Safety- Critical Industries

CodePeer has been qualified as a Verification Tool under DO-178B, a software standard for commercial airborne systems, automating a number of activities associated with that standard's objectives for software accuracy and consistency. CodePeer has also been qualified for CENELEC EN 50128. The CENELEC EN 50128 qualification material addresses boundary value analysis (detecting errors such as buffer overflow), control flow analysis (detecting errors such as unreachable code), and data flow analysis (detecting errors such as references to uninitialized variables).

Qualification material for both DO-178B/C and CENELEC EN 50128 is available as a product option.

3.4.3 How can CodePeer help your Software Project?

- Finds potential bugs and vulnerabilities early, when they are less expensive to correct
- Expedites code review and significantly increases the productivity of human review
- Detects and removes latent bugs when used retrospectively on existing code
- Reduces effort needed for safety or security certification
- Improves code quality
- Works on partially complete programs
- Exploits multi-core CPUs for efficiency and allows performance tuning based on memory and speed of developer’s machine

3.4.4 What makes the CodePeer approach unique?

CodePeer offers a number of advantages over other tools:

- Ease of use with GNAT Pro, so that no special setup is needed
- Helpful output such as the generation of subprogram summaries
- The ability to analyze a subprogram or a package in isolation: there is no need for a driver that gives a calling context, whether manually written or generated
- The ability to detect logic errors such as assigning to a variable that is never subsequently referenced or testing a condition that always evaluates to the same true or false value
- Automatic generation of both human-readable and machine-readable component specifications: preconditions and postconditions, inputs and outputs, heap allocations
- Automated code reviews
- Warnings ordered by ranking, so that more severe and likely errors are treated first, with ranking heuristics fully customizable by the user

3.5 Basic Static Analysis tools

3.5.1 ASIS, GNAT2XML

ASIS is a library 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.

ASIS, the Ada Semantic Interface Specification, is an international standard (ISO/IEC 15291:1995), and is designed to be compiler independent. Thus a tool that processes the ASIS representation of a program will work regardless of which ASIS implementation has been used. ASIS-for-GNAT is AdaCore's implementation of the ASIS standard, for use with the GNAT Pro Ada development environment and toolset.

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

Typical ASIS-for-GNAT 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 Standards Enforcement Tools
- Language Translators (e.g., to CORBA IDL)
- Quality Assessment Tools
- Source Browsers and Formatters
- Syntax Directed Editors

GNAT2XML provides the same information as ASIS, but allows users to manipulate it through an XML tree.

3.5.2 GNATmetric

The GNATmetric tool analyzes source code to calculate a set of commonly used industry metrics that allow 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.

3.5.3 GNATcheck

GNATcheck is a coding standard verification tool that is extensible and rule-based. It allows developers to completely define a coding standard as a set of rules, for example a subset of permitted language features. It verifies a program's conformance with the resulting rules and thereby facilitates demonstration of a system's compliance with certification standards such as CENELEC EN 50128 or DO-178B/C.

Key features include:

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

3.5.4 GNATstack

GNATstack is a software analysis tool that enables Ada/C/C++ software development teams to accurately predict the maximum size of the memory stack required to host an embedded software application.

The GNATstack tool 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 perform per-subprogram stack usage computation combined with control flow analysis.

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

This is a static tool in the sense that its computation is based on information known at compile time. It implies that when the tool indicates that the result is accurate then the computed bound can never overflow.

On the other hand, there may be situations in which the results will not be accurate (the tool will actually indicate this situation) because of some missing information (such as subprogram recursion, indirect calls, etc.). We provide the infrastructure to allow users to specify this missing call graph and stack usage information.

The main output of the tool is the worst-case stack usage for every entry point, together with the paths that lead to these stack needs. 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 certain regular expression).

The tool 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 we do not have any stack or call graph information.

- **Unbounded frames**. The tool displays all the subprograms that are reachable from any entry point with an unbounded stack requirements. The required stack size depends on the arguments passed to the subprogram.
- **Cycles**. The tool can detect all the cycles in the call graph.

The tool will allow the user to specify in a text file the missing information, such as the potential targets for indirect calls, stack requirements for externals calls, and user-defined bounds for unbounded frames.

3.6 GNATtest, GNATemulator and GNATcoverage

3.6.1 GNATtest

The GNATtest tool helps create and maintain a complete unit testing infrastructure for complex projects. Based on AUnit, it captures the simple idea that each visible subprogram 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.

See more at:

http://www.adacore.com/gnatpro/toolsuite/gnattest/

3.6.2 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, the GNATemulator tool 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 go far in replacing the final hardware during development for all sorts 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 the GNATemulator, do not pretend to be complete time-accurate target board simulators, and thus cannot be used for all aspects of testing, but do provide a very efficient, cost-effective way of executing the target code very early and very broadly in the development and verification process. They offer a practical compromise between a native environment that is too far from the actual target, and the final hardware that might not be available soon enough or in sufficient quantity.

3.6.3 GNATcoverage

GNATcoverage is a specialized tool that analyzes and reports program coverage. GNATcoverage allows coverage analysis of both object code (instruction and branch coverage), and Ada or C language source code (Statement, Decision and Modified Condition/Decision Coverage - MC/DC).

Unlike most current technologies, the tool works without requiring instrumentation of the application code. Instead, the code runs directly on an instrumented execution platform, such as GNATemulator, Valgrind on Linux, or on a real board monitored by a probe.

See [BOR 10] for more details on the underlying technology.

3.7 IDEs

3.7.1 GPS

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

Tools you can use

GPS's extensive navigation and analysis tools can generate a variety of useful information including call graphs, source dependencies, project organization, and complexity metrics, giving you a thorough understanding of your program at multiple levels. It allows you to interface with third-party Version Control Systems, easing both development and maintenance.

Robust, Flexible and Extensible

Especially suited for large, complex systems, GPS lets you import existing projects from other Ada implementations while adhering to their file naming conventions and retaining your directory organization. Through its multi-language capabilities you can also handle components written in C and C++. GPS is highly extensible; a simple scripting approach lets you plug in additional tools. It is also tailorable, allowing you to specialize various aspects of the program's appearance in the editor.

Easy to learn, easy to use

If you are a new user, you will appreciate GPS's intuitive 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. Experienced users will appreciate that GPS offers the necessary level of control for advanced uses; e.g. the ability to run command scripts. Anything you can do on the command line is achievable through the menu interface.

Remote Programming

Integrated into GPS, Remote Programming provides a secure and efficient way for programmers to access any number of remote servers running a wide variety of platforms while taking advantage of the power and familiarity of their local PC workstations.

GPS Benefits at a glance

- Management of complexity, through tools that provide specialized views of the program components and their interrelationships
- Ease of learning, through a platform-independent visual interface
- Automation of the program build process, through a project manager tool that offers complete control over switch settings, file location, etc.
- Ease of debugging, through a fully integrated visual debugger
- Support for configuration management, through an interface to 3rd-party version control systems

- Adaptability, through facilities that allow GPS to be extended or tailored
- Compatibility of new versions of GPS with older versions of GNAT Pro

3.7.2 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 development with some support for cross development. In both cases the Ada tools are tightly integrated.

3.7.3 GNATdashboard

GNATdashboard essentially serves as a one-stop control panel for monitoring and improving the quality of Ada software. It integrates and aggregates the results of AdaCore's various static and dynamic analysis tools (GNATmetric, GNATcheck, GNATcoverage, CodePeer, 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.

Features of GNATdashboard

- Provides a common interface to view code quality info and metrics such as:
    - Code complexity
    - Code coverage
    - Conformance
- Fits into a continuous integration environment
- Uses project files to configure, run and analyze the tools' output
- Integrates with the SQUORE and SonarQube platforms to visualize the results

Benefits of GNATdashboard

- Allows QA managers and project leads to understand their technical debt

- Allows teams to engage all developers to track and reduce technical debt
- Eliminates the need for manual input

3.8 QGen

QGen is a qualifiable and tunable code generation and model verification tool for a safe subset of Simulink® and Stateflow® models. It reduces the development and verification costs for safety-critical applications through qualifiable code generation, model verification, and tight integration with AdaCore's qualifiable simulation and structural coverage analysis tools.

QGen answers one core question: how can I decrease the verification costs when applying model-based design and automatic code generation with the Simulink® and Stateflow® environments? This is achieved by

- Selecting a safe subset of Simulink® blocks
- Ensuring high-performance and tunable code generation
- Relying on static analysis for upfront detection of potential errors, and
- Providing top-class DO-178B/C, CENELEC EN 50128 and ISO 26262 qualification material for both the code generator and the model verification tools. QGen also decreases tool integration costs by integrating smoothly with AdaCore's qualifiable compilation, simulation and structural coverage analysis products.

3.8.1 Support for Simulink® and Stateflow® models

QGen supports a wide range of features from the Simulink® and Statefow® environments, including more than 100 blocks, Simulink® signals and parameters objects and several Matlab® operations. The supported feature set from the Simulink® and Stateflow® environments has been carefully selected to ensure code generation that is amenable to safety-critical systems. MISRA Simulink® constraints can be optionally checked with QGen. Features that would imply unpredictable behavior, or that would lead to the generation of unsafe code, have been removed. The modelling standard enforced by QGen is then suitable for DO-178B/C, CENELEC EN 50128 and ISO 26262 development out-of-the-box.

3.8.2 Qualification material

Complete qualification material for QGen is planned; it will demonstrate compliance with the DO-178C standard at Tool Qualification Level 1 (TQL-1, equivalent to a Development Tool in DO-178B). This will make QGen the only code generator for Simulink® and Stateflow® models for which a TQL-1 qualification kit is available. The QGen qualification kit will show compliance with DO-330 (the DO-178C technology supplement on Model-Based Development) and include a Tool Qualification Plan, a Tool Development Plan, a Tool Verification Plan, a Tool Quality Assurance Plan and a Tool Configuration Management Plan; it will also include detailed Tool Operational Requirements, Test Cases and Test Execution Results.

DO-330 is compatible with CENELEC EN 50128 from the point of view of Railway Assessor. See [BOU 13] for more details.

3.8.3 Support for model static analysis

QGen supports the static verification that three kinds of issues are prevented: run-time errors, logical errors, and safety violations. Run-time errors, such as division by zero or integer overflow, may lead to exceptions being raised during system execution. Logical errors, for example a Simulink® “If” block condition that is always true, imply a defect in the designed model. And safety properties, which can be modeled using Simulink® Model Verification blocks, represent safety requirements that are embedded in the design model. QGen is able to statically verify all these properties and generate run-time checks as well if configured to do so.

3.8.4 Support for Processor-in-the-Loop testing

QGen can be integrated with AdaCore’s GNATemulator and GNATcoverage tools to support streamlined Processor-In- the-Loop (PIL) testing. The simulation of Simulink® models can be tested back-to-back against the generated code, which is cross-compiled and deployed on a GNATemulator installation on the user workstation. While conducting PIL testing, GNATcoverage can also perform structural coverage analysis up to MC/DC without any code instrumentation. Both GNATcoverage and GNATemulator have been already qualified in an operational context.

CHAPTER 4

ADACORE CONTRIBUTIONS TO THE SOFTWARE QUALITY ASSURANCE PLAN

4.1 Software Architecture (A.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 existence of some capabilities at the lower level may enable certain design decisions at a higher level. This table contains some hints of how that can be done.

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Defensive Programming	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. In addition, advanced static analysis tools such as CodePeer and SPARK help identifying pieces of code that should be protected by defensive code.
Fault Detection & Diagnosis	R	R	No	
Error Correcting Codes	-	-	No	
Error Detecting Codes	R	HR	No	
Failure Assertion Programming	R	HR	Yes	The **Ada** language allows formalizing assertions and contracts in various places in the code.
Safety Bag Techniques	R	R	No	

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Diverse Programming	R	HR	Yes	Using Ada along with another language can be used to contribute to the diverse programming argument.
Recovery Block	R	R	No	
Backward Recovery	NR	NR	No	
Forward Recovery	NR	NR	No	
Retry Fault Recovery Mechanisms	R	R	No	
Memorising Executed Cases	R	HR	No	
Artificial Intelligence - Fault Correction	NR	NR	No	
Dynamic Reconfiguration of software	NR	NR	No	
Software Error Effect Analysis	R	HR	No	
Graceful Degradation	R	HR	No	
Information Hiding	-	-	Yes	Information hiding is not recommended by the standard, as it makes data non-observable. In **Ada**, information encapsulation will be preferred.
Information Encapsulation	HR	HR	Yes	The **Ada** language provides the necessary features to separate the interface of a module from its implementation and enforce respect of this separation.
Fully Defined Interface	HR	M	Yes	The **Ada** language provides the necessary features to separate the interface of a module from its implementation and enforce respect of this separation.
Formal Methods	R	HR	Yes	**SPARK** can be used to formally define architecture properties, such as data flow, directly in the code and provide means to verify them.
Modelling	R	HR	Yes	**Ada** and **SPARK** allow defining certain

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
				modelling properties in the code and provide means to verify them.
Structured Methodology	HR	HR	Yes	Structured Methodology designs can be implemented with **Ada**.
Modelling supported by computer aided design and specification tools	R	HR	No	

4.2 Software Design and Implementation (A.4)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Formal Methods	R	HR	Yes	Component requirements and interfaces can can be written in the form of formal boolean properties, using the **Ada** or **SPARK** language. These properties are verifiable.
Modelling	HR	HR	Yes	**Ada** and **SPARK** allow defining certain modelling properties in the code and provide mean to verify them.
Structured methodology	HR	HR	Yes	Structured Methodology designs can be implemented with **Ada**.
Modular approach	M	M	Yes	A module can be represented as an Ada package, with a well-defined functionality, a clear external interface in the package spec, a private part to limit the visibility only to children packages, and a body containing the implementation which is not visible to any other module.
Components	HR	HR	Yes	A component can be defined as a set of **Ada** packages, 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 standard	HR	M	Yes	There are available references for the coding standard. Verification can be

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
				automated in different ways: The **GNAT** compiler can define base coding standard rules to be checked at compile-time, **GNATcheck** implements a wider range of rules, and **GNAT2XML** can be used to develop specific coding rules.
Analyzable programs	HR	HR	Yes	The **Ada** language provides 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. **CodePeer** allows making an assessment of program analyzability during its development. For higher levels, the use of **SPARK** ensures that the subset of the language used is suitable for the most rigorous analyses.
Strongly typed programming language	HR	HR	Yes	**Ada** is a strongly typed language.
Structured Programming	HR	HR	Yes	**Ada** supports all the usual paradigms of structured programming. In addition to these, **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.
Programming language	HR	HR	Yes	**Ada** can be used for most of the development, with potential connection to other languages such as C or assembly.
Language subset	-	HR	Yes	The **Ada** language is designed to be easily subsetted, possibly under the control of specific run-times, **GNATcheck**, or with **SPARK**. Another possibility is to follow the recommendations made by the Guide for the Use of the Ada Programming Language in High Integrity Systems
Object-oriented programming	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	HR	HR	Yes	**Ada** supports all the usual paradigms of

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
programming				procedural programming.
Metaprogramming	R	R	No	

4.3 Verification and Testing (A.5)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Formal proofs	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 regarding these contracts.
Static analysis	HR	HR	Yes	See table A.19
Dynamic analysis and testing	HR	HR	Yes	See table A.13
Metrics	R	R	Yes	**GNATmetric** can retrieve metrics, such as code size, comment percentage, cyclomatic complexity, unit nesting, and loop nesting. These can then be compared with standards.
Traceability	HR	M	No	
Software error effect analysis	R	HR	Yes	**GPS** supports code display and navigation. **CodePeer** can identify likely errors locations in the code. This supports potential software error detection and analysis throughout the code.
Test coverage for code	HR	HR	Yes	See table A.21
Functional / black-box testing	HR	HR	Yes	See table A.14
Performance testing	HR	HR	No	
Interface testing	HR	HR	Yes	The strong typing provided by **Ada** together with function contracts provide increased assurance to demonstrate that the software interfaces do not contain any errors at

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
				the software level. This can help improve integration testing.

4.4 Integration (A.6)

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

4.5 Overall Software Testing (A.7)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Performance Testing	HR	M	Yes	Stack consumption can be statically studied using the **GNATstack** tool.
Functional and Black-box testing	HR	M	Yes	**GNATtest** can generate a testing framework for testing.
Modelling	R	R	No	

4.6 Software Analysis Techniques (A.8)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Static Software Analysis	HR	HR	Yes	See table A.19
Dynamic Software Analysis	R	HR	Yes	See table A.13 / A.14
Cause Consequence Diagrams	R	R	No	

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Event Tree Analysis	R	R	No	
Software Error Effect Analysis	R	HR	Yes	**GPS** supports code display and navigation. **CodePeer** can identify likely error locations in the code. These tools support both detection of potential software errors and analysis throughout the code.

4.7 Software Quality Assurance (A.9)

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	SIL 2	SIL 3/4	Covered	Comment
Accredited to EN ISO 9001	HR	HR	No	
Compliant with EN ISO 9001	M	M	No	
Compliant with ISO/IEC 90003	R	R	No	
Company Quality System	M	M	No	
Software Configuration Management	M	M	No	
Checklists	HR	M	No	
Traceability	HR	M	No	
Data Recording and Analysis	HR	M	Yes	The data produced by tools can be written to files and put in configuration management systems.

4.8 Software Maintenance (A.10)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Impact Analysis	HR	M	Yes	The **CodePeer** tool contributes to identifying the impact of a code change between two baselines, from the static analysis point of view.
Data Recording and Analysis	HR	M	Yes	AdaCore tools are driven from the command line and produce result files including the date and version of the tool used.

4.9 Data Preparation Techniques (A.11)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Tabular Specification Methods	R	R	Yes	Tables of data can be expressed using the Ada language, together with type-wide contracts (predicates or invariants).
Application specific language	R	R	No	
Simulation	HR	HR	No	
Functional Testing	M	M	No	
Checklists	HR	M	No	
Fagan inspection	HR	HR	No	
Formal design reviews	HR	HR	Yes	**GPS** can display code and navigate through the code as a support for walkthrough activities.
Formal proof of correctness	-	HR	Yes	When contracts on arrays are expressed within the **SPARK** subset, the correctness of these contracts can be formally verified.
Walkthrough	R	HR	Yes	**GPS** can display code and navigate through the code as a support for walkthrough activities.

4.10 Coding Standards (A.12)

There are available references of 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. GNATcheck implements a wider range of rules. GNAT2XML can be used to develop specific coding rules.

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Coding Standard	HR	M	Yes	**GNATcheck** allows implementing and verifying a coding standard.
Coding Style Guide	HR	HR	Yes	**GNATcheck** allows implementing and verifying a coding style guide.
No Dynamic Objects	R	HR	Yes	**GNATcheck** can forbid the use of dynamic objects.
No Dynamic Variables	R	HR	Yes	**GNATcheck** can forbid the use of dynamic variables.
Limited Use of Pointers	R	R	Yes	**GNATcheck** can forbid the use of pointers or force justification of their usage.
Limited Use of Recursion	R	HR	Yes	**GNATcheck** can forbid the use of recursion or force justification of their usage.
No Unconditional Jumps	HR	HR	Yes	**GNATcheck** can forbid the use of unconditional jumps.
Limited size and complexity of Functions, Subroutines and Methods	HR	HR	Yes	**GNATmetric** can compute complexity and **GNATcheck** can report excessive complexity.
Entry/Exit Point strategy for Functions, Subroutines and Methods	HR	HR	Yes	**GNATcheck** can verify rules related to exit points.
Limited number of subroutine parameters	R	R	Yes	**GNATcheck** can limit the number of parameters for subroutines and report when that number is exceeded.
Limited use of Global Variables	HR	M	Yes	**GNATcheck** can flag global variable usage and enforce their justification. **SPARK** can enforce documentation and verification of functions' side effects, including usage of global variables. GPS's cross referencing capabilities

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
				allow the analysis and verification of the usage of global variables.

4.11 Dynamic Analysis and Testing (A.13)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Test Case Execution from Boundary Value Analysis	HR	HR	Yes	**GNATtest** can generate and execute a testing framework for an actual test written by developers from requirements.
Test Case Execution from Error Guessing	R	HR	No	
Test Case Execution from Error Seeding	R	HR	No	
Performance Modelling	R	HR	No	
Equivalence Classes and Input Partition Testing	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	R	HR	Yes	See table A.21

4.12 Functional/Black Box Test (A.14)

GNATtest can generate and execute a testing framework - actual test being written by developers from requirements.

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Test Case Execution from Cause Consequence Diagrams	-	R	No	
Prototyping/ Animation	-	R	No	

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Boundary Value Analysis	R	HR	Yes	**GNATtest** can be used to implement tests coming from boundary value analysis.
Equivalence Classes and Input Partitioning Testing	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	R	R	No	

4.13 Textual Programming Language (A.15)

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	The **GNAT Pro** compiler supports C and C++
PL/M	R	NR	No	
BASIC	NR	NR	No	
Assembler	R	R	No	
C#	R	R	No	
Java	R	R	No	
Statement List	R	R	No	

4.14 Modelling (A.17)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Data Modelling	R	HR	Yes	**Ada** allows modelling data

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
				constraints, in the form of subtype predicates.
Data Flow Diagram	R	HR	Yes	**SPARK** allows defining data flow dependences at subprogram specification.
Control Flow Diagram	R	HR	No	
Finite State Machine or State Transition Programs	HR	HR	No	
Time Petri Nets	R	HR	No	
Decision/Truth Tables	R	HR	No	
Formal Methods	R	HR	Yes	**Ada** and **SPARK** allow defining formal properties on the code that can be verified by the **SPARK** toolset.
Performance Modelling	R	HR	No	
Prototyping/Animation	R	R	No	
Structure Diagrams	R	HR	No	
Sequence Diagrams	R	HR	No	

4.15 Performance Testing (A.18)

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

4.16 Static Analysis (A.19)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Boundary Value Analysis	R	HR	Yes	**CodePeer** can computes boundary values for variables and parameters from the source code. **CodePeer** and **SPARK** can provide various verification looking at potential values and boundary values of variables. Detected errors include attempts to dereference a variable that could be null, values outside the bounds of an Ada type or subtype, buffer overflow, numeric overflow or wraparound, and division by zero. CodePeer and SPARK can also help confirming expected boundary values of variables and parameters coming from the design.
Checklists	R	R	No	
Control Flow Analysis	HR	HR	Yes	**CodePeer** 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 compute the maximum amount of memory used in stacks looking at the control flow. More generally, **GPS** provides visualization for call graphs and call trees.
Data Flow Analysis	HR	HR	Yes	**CodePeer** and **SPARK** can detect suspicious and potentially incorrect data flow, such as variables being read before they're written (uninitialized variables), values that are written to variables without being read (redundant assignments or variables that are written but never read).
Error Guessing	R	R	No	
Walkthroughs/Design Reviews	HR	HR	Yes	**GPS** can display code and navigate through the code, thus supporting walkthrough activities.

4.17 Components (A.20)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Information Hiding	-	-	No	Information hiding is not recommended by the standard, as it makes data non-observable. In Ada, information encapsulation is preferred methodologically, but Ada still allows access through child units (especially private child units) thus addressing the objection raised by the standard.
Information Encapsulation	HR	HR	Yes	**Ada** provides the necessary features to separate the interface of a module from its implementation, and to enforce this separation semantically.
Parameter Number Limit	R	R	Yes	**GNATcheck** can limit the number of parameters for subroutines and report violations.
Fully Defined Interface	HR	M	Yes	**Ada** offers many features to complete interface definition, including behavior specification.

4.18 Test Coverage for Code (A.21)

Technique/Measure	SIL 2	SIL 3/4	Covered	Comment
Statement	HR	HR	Yes	**GNATcoverage** provides statement-level coverage capabilities.
Branch	R	HR	Yes	**GNATcoverage** provides branch-level coverage capabilities.
Compound Condition	R	HR	Yes	**GNATcoverage** provides MC/DC coverage capabilities, which can be used as an alternative to Compound Conditions.
Data Flow	R	HR	No	
Path	R	NR	No	

4.19 Object Oriented Software Architecture (A.22)

Technique/Measure	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 with **Ada**.
Object Oriented Detailed Design	R	HR	Yes	See table A.23

4.20 Object Oriented Detailed Design (A.23)

Technique/Measure	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** supplies an explicit notation for overriding methods, which can be enforced by GNAT compiler switches.
Multiple inheritance used only for interface classes	HR	HR	Yes	**Ada** only allows multiple inheritance from interfaces.
Inheritance from unknown classes	-	NR	No	

CHAPTER 5

TECHNOLOGY USAGE GUIDE

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, such as code reviews, and automatic analysis, ranging from code verification performed by the compiler to formal proof.

Examples of these language features include:

- type (and subtype) ranges and predicates
- parameter modes and subprogram contracts
- encapsulation
- minimal set of implementation-dependent or undefined behaviors

Tools such as **GNATmetric** and **GNATcheck** allow monitoring the complexity and quality of the code and identifying potentially problematic situation. These are done by using such methods as basic code size metrics, cyclomatic complexity, and coupling analysis.

The **CodePeer** static analysis tool looks for 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 **CodePeer** during development allows spotting locations in the code that are potentially too complex and provides information on which aspects need to be improved.

The **SPARK** language is more extensive in analyzing programs, aiming at full correctness proofs. It structurally forbids unanalyzable features and constructs. Such proofs can only be performed if the code if clear and well designed. If not, it’s unlikely that a proof will be constructed, even on code that could potentially be correct. Using **SPARK** during development ensures maximum analyzability of the code from an automatic point of view.

During code review phases, **GPS** 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 the function at the limits and boundaries 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 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 input and output of a subprogram, in the form of a partitioning of the input domain:

```
function Compute (I : Integer) return Integer
  with Contract_Cases => (I = Integer'First => Compute'Result = -1,
                          I = Integer'Last => Compute'Result = 1,
                          others => I - 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 *I* 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 "D.18 Equivalence Classes and Input Partitioning".

Another possibility is to use **CodePeer** 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, **CodePeer** 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 **CodePeer** 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.

CodePeer can also 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 can perform similar analysis for freedom of exceptions, 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. This includes unreachable code and useless tests in the code, such as conditions that are always true.

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

CodePeer contributes to control flow analysis by identifying unreachable code as well as conditions that are 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** toolchain can be configured to detect uninitialized variables at run-time through the use of the pragma Initialize_Scalar. When using this pragma, all scalars are automatically initialized to either an out-of-range value (if such exist) or to a very large number. This significantly improves detection at test time.

CodePeer and **SPARK** can detect suspicious and potentially incorrect data flows, such as variables 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 allows reaching higher levels of confidence.

SPARK allow going much further, 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 any side effects
- verification that all dependencies between inputs and outputs are specified
- verification that the implemented dataflow corresponds to the one specified

Let's take one example to illustrate the above:

```
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;
   if A = 0 then
      R2 := B + C;
   else
      R2 := B - C;
   end if;
end Compute;
```

In the above *Depends* aspect, *R1* is required to be computed from *A* and *B* and *R2* from *B* and *C*. However, in the code, *R2* depends on the result of the condition "A = 0", so its value is actually computed from *A*, *B* and *C*, and not just *B* and *C*. Interestingly, formal proof detects such incorrect code whether or not branches are present:

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

This would produce similar results. Similar deductions can be made in presence of a call, assuming that a procedure should have an effect (which is reasonable). Let’s take the example of a logging call that we might forget in the code. In **SPARK**, side effects are documented. We would probably have a global state for it, let’s call it *Screen* in this example:

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

Again, slightly modifying our section of code:

```
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;
```

You can see that the data flow does not correspond to the specification: Compute should declare the fact that it modifies Screen. So the incorrect code is detected. Similarly to the earlier case, it’s worth noting that this incorrect code is detected even in the absence of a branch, making this a useful complement to structural code coverage in many cases.

5.5 Defensive Programming (D.14)

Defensive programming is about reducing anomalous control flow, data flow, or data values and reacting to these if necessary.

Ada's strong typing will structurally remove 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 subprogram specification, through preconditions, postconditions or predicates.

Beyond this, **Ada** provides specific support for a subset of what's specified in the D.14 annex. **CodePeer** 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, **GNAT2XML**.

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;
V : Percent;
-- [...]
V := X + Y; -- raises an exception if X + Y is out of range
```

The developer can provide exception handlers to respond to potential exceptions. Alternatively, it's possible to write explicit verification in the code to ensure that the expression is within its bounds; for example:

```
V1 : Integer;
V2 : Percent;
-- [...]

if V1 in Percent then
   V2 := V1;
```

```
end if;
```

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

Note that all these checks can be deactivated, for example once thorough testing or formal proof has been performed.

5.5.2 Data should be dimension checked

The GNAT compiler provides an implementation-defined aspect and package for dimensional consistency analysis, which ensures that variables are properly typed according to their dimensions. The system is implemented on top of the 7 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 time per distance. 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 modes must be specified in parameter specifications and are 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 as mode *in out* and is modifiable. The compiler will produce an error for an attempted modification of *in* parameters and detect (and warn) when an *in out* parameter is not modified and so could have been passed as *in*. For example:

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

```
end P;
```

5.5.4 Functions should treat all parameters as read-only

The initial Ada standard completely prohibited non-read-only function parameters. Later versions allow it, but the behavior can be easily reverted through a GNATcheck rule. The SPARK Ada subset does forbid functions with writable parameters.

5.5.5 Literal constants should not be write-accessible

Ada implements many ways to define literals, through either constants or enumerations, for example:

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

These are read-only as per language definition. Even if a literal value (such as a string literal) or a large constant (such as a record or array aggregate) is passed by reference, it will not be write accessible in the called subprogram.

5.5.6 Using CodePeer and SPARK to drive defensive programming

CodePeer 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 such defensive code. Let’s take one example:

```
procedure P (V : Integer);
procedure P (V : Integer) is
begin
   -- [...]
   Some_Array (V) := …
   -- [...]
end P;
```

In the above code, there’s a use of V as an index of Some_Array. CodePeer will detect the potential for a run-time error that the code needs to be protected against. This protection can either be in the form of specific tests, as show below:

```
procedure P (V : Integer);
procedure P (V : Integer) is
begin
   if V not in Some_Array'Range then
      return;
   end if;
   -- [...]
   Some_Array (V) :=
   -- [...]
end P;
```

or in the form of a precondition, to issue the error at call time and therefore protect this very subprogram from the incorrect condition:

```
procedure P (V : Integer)
  with Pre => V in Some_Array'Range;

procedure P (V : Integer) is
begin
   -- [...]
   Some_Array (V)
   -- [...]
end P;
```

The main difference between **CodePeer** and **SPARK** in the above example is that **CodePeer** 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 will test for all potential run-time errors.

In general, the preferred Ada style is to use contracts instead of defensive code since a precondition makes the requirement more obvious to the human reader (compared to a test in the middle of the procedure body) and also can be treated as a condition with a run-time check, a condition to be verified statically (SPARK), or a comment to the human reader.

5.6 Coding Standards and Style Guide (D.15)

Coding standard can be defined using a combination of predefined rules (using GNAT options and GNATcheck rules) and user-defined rules using either the ASIS API or XML-based checks on the output of GNAT2XML.

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

This technique is about partitioning the various potential inputs of subprograms and creating a testing and verification strategy based on this partitioning.

GNAT can provide support for specifying this partitioning at the source code level. The partition is a list of conditions of inputs with their associated expected output, verifying the following criteria:

- The full spectrum 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 partition 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.

5.8 Failure Assertion Programming (D.24)

Ada offers a large choice of assertions that can be defined in the code. They start with arbitrary verification within a sequence of statements:

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

Pre-conditions and post conditions can be defined on subprograms:

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

Predicates and invariants 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 technology allows fine control over which contracts need to remain and which need to be removed if the architecture requires some of them to be in the code after deployment. The contracts can be used by the static analysis and formal proof tools as well. CodePeer uses these 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, it's used as a way to provide the tool with additional information on the code behavior. SPARK will be able to go further and prove their correctness.

5.9 Formal Methods (D.28)

When using the SPARK language, formal methods can be used to define and check certain architectural properties, in particular with regard to 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 is indeed present but no others. *G* is defined as being evaluated depending on the values of *X* and *Y*. Again, **SPARK** will verify that the variable relationships are fully defined and 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 verification 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 and 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 logic of the first order (quantifiers). These properties can be expressed in the form of subprogram preconditions, postconditions, type invariants and subtype predicates. For example:

```
-- P must have input V 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;

-- This array is always sorted
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.

5.10 Impact Analysis (D.32)

Identifying the effect of a change on entire software component requires the combination of various techniques, including reviews, testing and static analysis. **CodePeer** has specific features to identify the impact of a change from the point of view of potential run-time errors. It can establish a baseline with regard to potential failure analysis and filter only the potential problems that have been introduced or fixed following a change in the code.

GPS can provide call graphs and call trees, revealing how a function is called in the software. This can be directly used in impact analysis.

5.11 Information Encapsulation (D.33)

Information hiding consists of making data unreachable. This is not considered good practice in EN 50128 as it makes diagnostic very difficult. Instead, Ada offers means to encapsulate the information in the following ways:

Proxies through exported variables

```
package body Data is
   G : Integer
      with Convention => C, External_Name => "var_g";
end Data;
```

Mapping of data in memory

```
package body Data is
   G : Integer
      with Address => 16#0000_56FF#;
end Data;
```

Use of accessors or modifiers

```
package Data is
   function Get_G return Integer;
   procedure Set_G (Val : in Integer);
end Data;

package body Data is
   G : Integer;
   function Get_G return Integer is
   begin
      return G;
   end Get_G;

   procedure Set_G (Val : in Integer) is
   begin
      G := Val;
   end Set_G;
end Data;
```

Protection of data through encapsulation, as explained below.

Like many other structured programming languages, **Ada** allows separating the user environment from the implementer environment. However, the granularity is different: although most languages rely on object-oriented patterns to perform this separation, Ada does it at the package (component) level.

Types can be encapsulated as a whole, whether they're implemented through classes or not. The following example demonstrates a small package creating an abstract counter, along with getters and setters:

```
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 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;

package body Counters is
   procedure Increment (C : in out Counter) is
   begin
```

```
      C := C + 1;
   end Increment;

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

5.12 Interface Testing (D.34)

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

- parameter passing modes
- pre and post conditions
- input partitioning
- typing

These are each described in other section 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.

5.13 Language Subset (D.35)

The **Ada** language has been designed to be easily subsetted. In its core definition, it defines a number of restrictions that can be applied, deactivating certain features of the language. **GNAT** run-times, such as the so-called Zero-Footprint [GNA 01] (no run-time component) or Ravenscar [BUN 04; MCC 11] (tasking subset) provide other natural subsets of the language, which have direct implications in terms of portability, determinism and safety.

SPARK is another natural Ada language subset, constraining the language to a formally analyzable subset (for example, no aliasing, no pointers, and no exceptions).

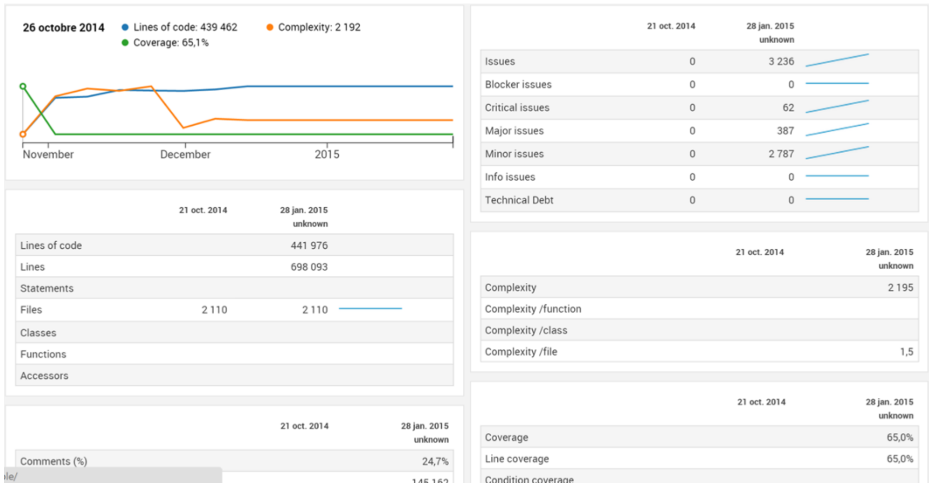
To go one step further, **GNATcheck** offers a number of features to verify that specific constructs are not present in the code and therefore that the code indeed complies with that subset.

5.14 Metrics (D.37)

GNATmetric computes 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 accessed through Python or SQL.

These metrics are eventually pushed into various interfaces, such as the SonarQube UI, shown below.



	21 oct. 2014	28 jan. 2015 unknown
Lines of code		441 976
Lines		698 093
Statements		
Files	2 110	2 110
Classes		
Functions		
Accessors		

	21 oct. 2014	28 jan. 2015 unknown
Comments (%)		24,7%

	21 oct. 2014	28 jan. 2015 unknown
Issues	0	3 236
Blocker issues	0	0
Critical issues	0	62
Major issues	0	387
Minor issues	0	2 787
Info issues	0	0
Technical Debt	0	0

	21 oct. 2014	28 jan. 2015 unknown
Complexity		2 195
Complexity /function		
Complexity /class		
Complexity /file		1,5

	21 oct. 2014	28 jan. 2015 unknown
Coverage		65,0%
Line coverage		65,0%
Condition coverage		

5.15 Modular Approach (D.38)

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

Ada provide native features to define groups of packages that have strong coupling, in particular through public and private child packages. In addition, the GNAT technology provides the notion of a project, which characterizes how a given set of source files will be built (tool switches, object directory, etc.). These constructs can be used to defined a tool-supported notion of “component” or “subsystem” at the software level.

A typical example is a complex system that needs to be spread across several packages. Let’s say we have one package, *Communication* and another package, *Interfaces*, that are contributing to the implementation of a signaling protocol. In Ada, it’s possible to design this in three (or more) distinct files in the following way:

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

The two private packages are defined in external 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. This is one of the several Ada features that allow designing application with strong internal coupling.

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

At a coarser granularity, packages (i.e. their source files) can be grouped together into a project, with a clear interface given by the name of the associated project file. An application architecture can be defined as a combination of project files, with various kinds of relationships between the projects.

5.15.2 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 children will be accessible to users.

5.15.3 Subprograms shall have a single entry and single exit only

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

5.15.4 Modules shall communicate with other modules via their interface

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

5.15.5 Module interface shall be fully documented

Although this is mostly the responsibility of the user, it should be noted that 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, of course, only the part of the documentation that can be expressed through boolean properties.

5.15.6 Interface shall contain the minimum number of parameter necessary

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

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

GNATcheck allows specifying a maximum number of parameters per subprogram.

5.15.18 Unit Proof and Unit Test

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

SPARK performs modular formal verification: it proves the post condition of a subprogram according to its own precondition and the precondition and postconditions of its callees (ie, those subprograms that it calls) whether or not these callees are themselves proven.

For a complete, 100%, proof, all the subprograms of an application need to be formally proven. But in situations where this is not possible, one subset can be proven and the others can be assumed.

These assumptions can then be verified using traditional testing methodology, allowing for a hybrid test / proof verification system.

5.16 Process Simulation (D.42)

The Matlab® Simulink® environment allows the development of a mathematical simulation of the control loop of a program. There are traditionally two parts in these environments, a simulation model and a control loop which is aimed at being embedded in the final application.

AdaCore has developed a Simulink® code generator, QGen, qualifiable with regard to the behavior of this simulation. In other words, the behavior observed during the simulation phase is identical to the behavior of the code once produced for the final target, ensuring the accurate representation and relevance of early simulation stages.

5.17 Strongly Typed Programming Languages (D.49)

Ada is a strongly typed language, which translates both into 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, in creating two types, *Kilometer* and *Miles*, the compiler will not allow mixed operation in the absence of explicit conversion by the user. 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 types ranges and predicates will be verified at specific points in the application, allowing early detection of inconsistencies.

5.18 Structure Based Testing (D.50)

AdaCore provides three tools to support structure based testing:

GNATtest is a unit testing framework generator. It takes Ada specifications and generates a test 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 or 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 CENELEC EN 50128 requires compound statements, Modified Condition/Decision Coverage (MC/DC) is usually accepted as a means of compliance.

5.19 Structured Programming (D.53)

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

5.20 Suitable Programming Languages (D.54)

Ada is referred to as "Highly Recommended" in the list of programming languages. Some features may however not be suitable for the highest level of software safety. In order to reach those, the language can be subsetted, see "D.35 Language Subset".

One of the advantage of the **Ada** language is that it is precisely defined as an international standard, ISO/IEC 8652. This document defines the expected behavior as well as implementation-defined behavior and includes specifications for the standard Ada libraries.

5.21 Object Oriented Programming (D.57)

Ada offers all the usual constructs for object-oriented programming. In addition to these, the Liskov Substitution Principle can be verified through class-wide contracts and SPARK formal verification, allowing the verification of class hierarchy consistency and safety of dispatching operations.

Ada is particularly well suited to be used in conjunction with safety critical applications as it allows instantiating objects on the stack. For example:

```
O : Some_Type'Class := Make_Some_Type;
```

In the above code, *O* is a polymorphic object that can be initialized with any value of the class *Some_Type*. It's allocated on the stack at initialization time. The booklet [GNA 13] provide additional information on how to use object-oriented features in certified context.

5.22 Procedural Programming (D.60)

Ada implements all the usual features of procedural programming languages.

5.23 Domain Specific Languages (D.71)

Simulink® and StateFlow® can be used as a domain specific language, using **QGen** to generate **SPARK** or MISRA-C code. The generator can be certified at the SIL4 level. In this context, development and test activities can be done at the model level and components tests can be directly deduced from simulation cases. This allows removing specific test-level components that usually need to be developed for the code. Code generated by **QGen** can be automatically exercised through the simulation cases to verify structural code coverage.

QGen provides verification for safe subsets for Simulink® and StateFlow® that can be used as off-the-shelf modelling standards.

CHAPTER 6

TECHNOLOGY ANNEX

6.1 Ada

6.1.1 Qualification

Although there is no qualification of a language per se, the Ada language is standardized by ISO as IEC/ISO 8652. Compilers and tools have a well-defined and official reference manual which precisely describes the expected behavior and the permitted implementation defined characteristics.

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 SPARK

6.2.1 Qualification

The SPARK analysis tool can be qualified as a T2 tool.

6.2.2 Annex D References

SPARK 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

6.3 GNAT

6.3.1 Qualification

The GNAT Pro compiler is qualified at the T3 level. AdaCore can provide a document attesting to various aspects such as service history, development standard, and testing results. That document has already been used successfully in past certification activities.

Notwithstanding the above, it’s recognized that compilers are usually not free of bugs and that bugs can be detected after a compiler version has been chosen. As per the requirements stated in 6.7.4.11 however, a “bug-fix” version of the compiler cannot be deployed in the absence of specific justification. As a specific service on a specific version of the technology, AdaCore offers critical problem fixes as well as a detailed description of the changes, allowing customers to integrate updated versions of the compiler with their process.

6.3.2 Run-Time Certification

High-Integrity run-times have been certified at the SIL-3/4 level.

6.3.3 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

6.4 CodePeer

6.4.1 Qualification

CodePeer can be qualified as a T2 tool. It has a long industrial track record in several domains and is qualified against other standards as well, such as DO-178B/C as a verification tool (TQL-5).

6.4.2 Annex D References

CodePeer 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

6.5 Basic Static Analysis tools

6.5.1 Qualification

These tools can be qualified as T2 tools. Some of them, such as GNATcheck, have been qualified under other standards as well, such as DO-178B/C as a verification tool (TQL-5).

6.5.2 Annex D References

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

6.6 GNATtest, GNATemulator and GNATcoverage

6.6.1 Qualification

These tools can be qualified as T2 tools. Some of them, such as **GNATcoverage**, have been qualified under other standards as well, such as DO-178B/C as a verification tool (TQL-5).

6.6.2 Annex D References

- D.50 Structure Based Testing

6.7 QGen - Simulink® Code Generator

6.7.1 Qualification

This tool can be qualified as a T3 tool or certified at the SIL3/4 levels.

6.7.2 Annex D References

- D.42 Process Simulation
- D.71 Domain Specific Languages

REFERENCES

[ADA 12] ISO/IEC, *Ada Reference Manual*, 2012

[BAR 13] John Barnes and Ben Brosgol, *Safe and Secure Software, an invitation to Ada 2012*, AdaCore, 2013

[BAR 14] John Barnes, *Programming in Ada 2012* Cambrigde University Press, 2014

[BAR 15] John Barnes, Ben Brosgol: *Safe and Secure Software, an invitation to Ada 2012* AdaCore, 2015

[BLA 11] Paul E. Black, Michael Kass, Michael Koo & Elizabeth Fong, *Source Code Security Analysis Tool Functional Specification*, NIST, 2011

[BOU 07] Jean-Louis Boulanger, Walter Schön, *Assessment of Safety Railway Applications*, ESREL, 2007.

[BOU 13] Jean-Louis Boulanger, *Tool qualification for the CENELEC EN 50128:2011*, 2013

[BOU 15] Jean-Louis Boulanger, *CENELEC 50128 and IEC 62279 standards*, ISTE-WILEY, London, 2015.

[BOR 10] Matteo Bordin, Cyrille Comar, Tristan Gingold, Jérôme Guitton, Olivier Hainque, Thomas Quinot, *Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework*, ERTS, 2010

[BUN 04] Alan Burns, Brian Dobbing and Tullio Vardanega: *Guide for the use of the Ada Ravenscar Profile in high integrity systems* Ada Letters, June 2004

[CEN 00] CENELEC, *EN 50126, Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*, 2000

[CEN 03] CENELEC, *EN 50129 Railway applications. Communication, signalling and processing systems. Safety related electronic systems for signalling*, 2003

[CEN 01] CENELEC, *EN 50128, Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems*, 2001

[CEN 11] CENELEC, *EN 50128, Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems*, 2011

[GNA 01] AdaCore, *GNAT Pro User's Guide Supplement for GNAT Pro Safety-Critical and GNAT Pro High-Security*

[GNA 13] AdaCore, *High-Integrity Object-Oriented Programming in Ada*, 2013

[MCC 11] John W. McCormick, Frank Singhoff, Jérôme Hugues, *Building Parallel, Embedded, and Real-Time Applications with Ada*, Cambridge University Press, 2011

[MCC 15] John W. McCormick and Peter C. Chapin, *Building High Integrity Applications with SPARK*, Cambridge University Press, 2015

[WWW 01] http://www.adacore.com/gnatpro-safety-critical/rail/