AdaCore technologies

for CENELEC EN 50128 2011

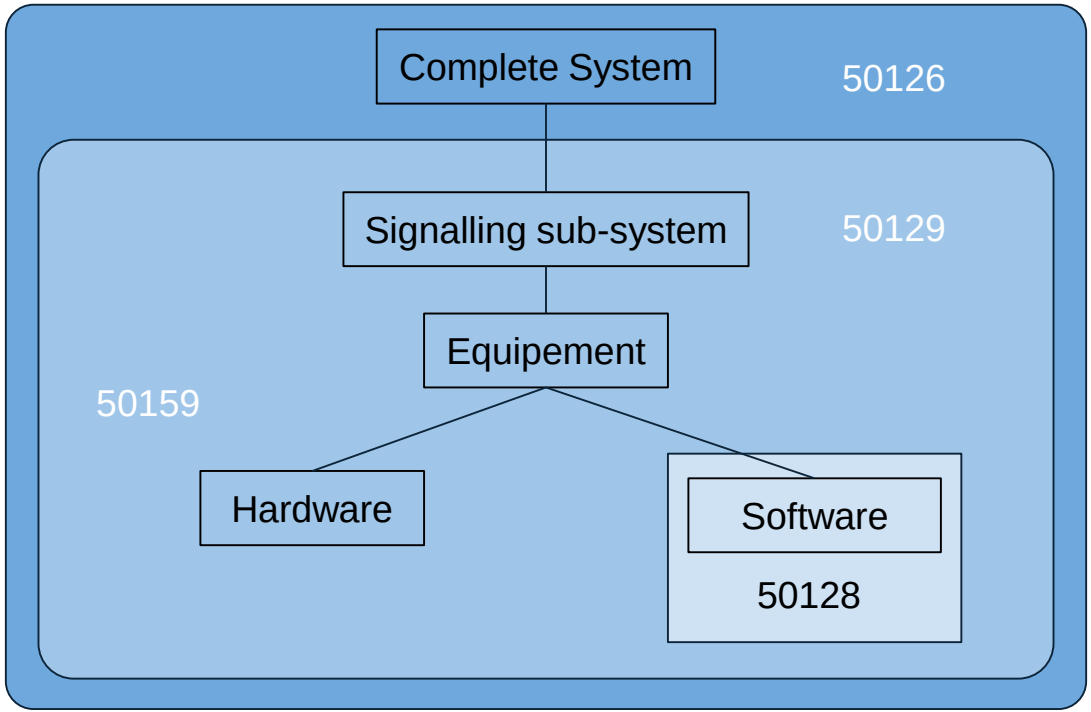
Eric Perlade
Technical Account Manager

RSSRail 2017

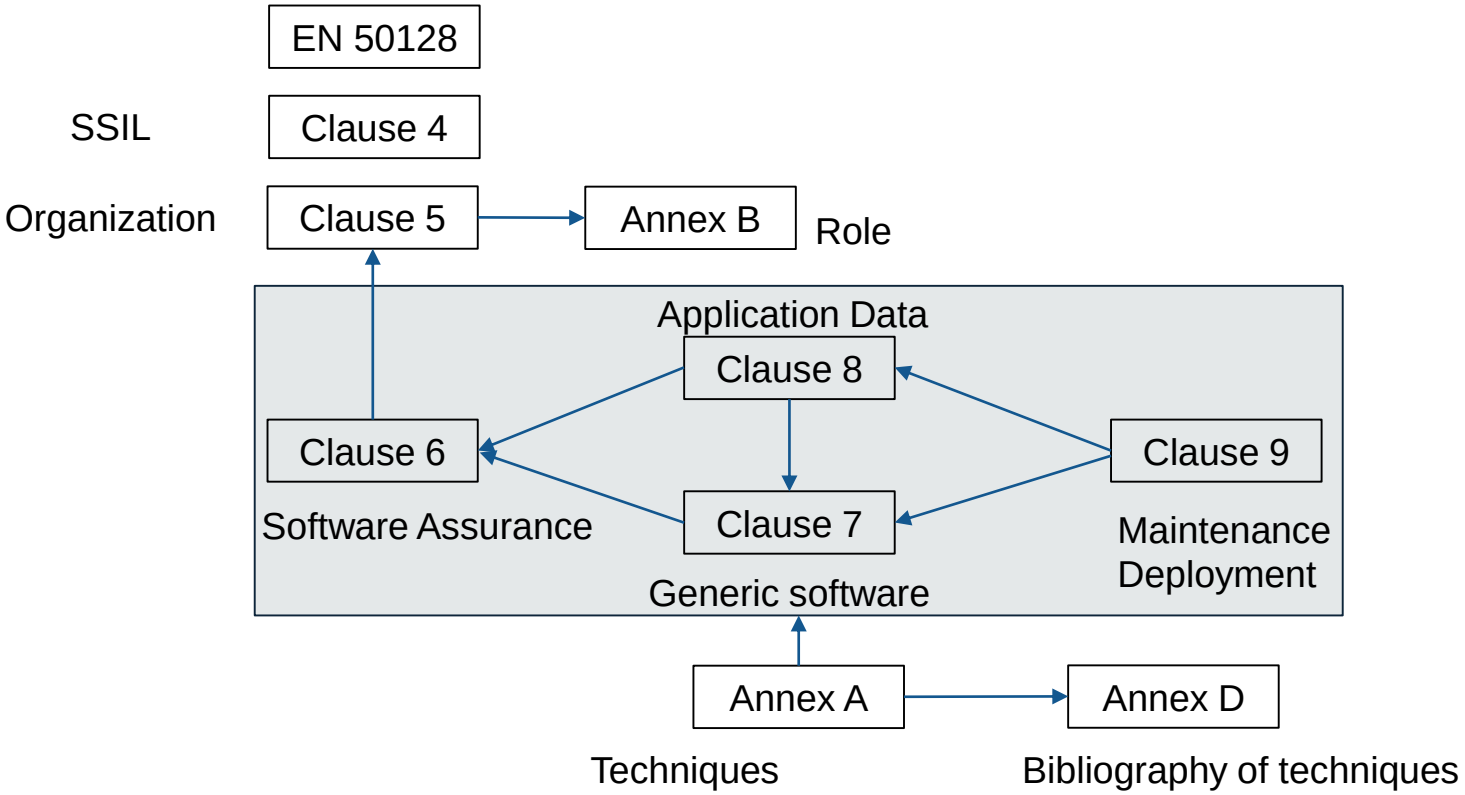
CENELEC EN 50128:2011

CENELEC EN 50128

- **Main standards applicable to railway systems**



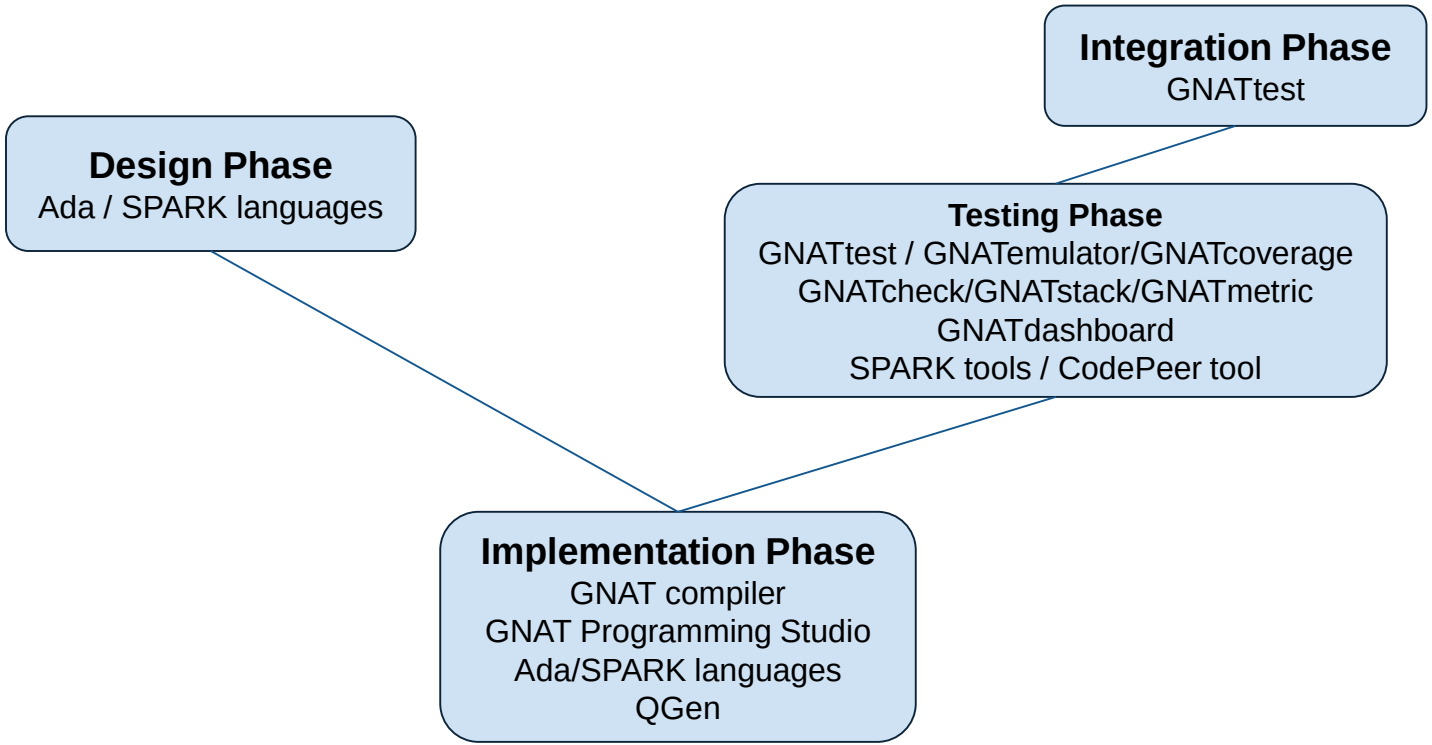
Structure of CENELEC 50128:2011



AdaCore tools to support EN 50128

- Ada 2012 language
- SPARK 2014 language and verification toolset performing formal proof and verification
- GNAT compiler
- CodePeer - static analysis tool that identifies potential run-time errors
- GNATmetric - metric computation tool
- GNATcheck - coding standard checker
- GNATdashboard - metric integration and management platform
- GNATtest - testing framework generator
- GNATemulator - processor emulator
- GNATcoverage - structural code coverage checker

Contributions of AdaCore tools to the V cycle



AdaCore contributions to the

Software Quality Assurance Plan

A.3 Software Architecture

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	Comment
Defensive Programming	HR	HR	Ada language + CodePeer + SPARK
Failure Assertion Programming	R	HR	The Ada language => assertions and contracts
Diverse Programming	R	HR	Ada <=> C / C++
Information Hiding	-	-	Ada language => information encapsulation
Information Encapsulation	HR	HR	The Ada language enforce separation between interface of a module and implementation
Fully Defined Interface	HR	M	The Ada language enforce separation between interface of a module and implementation
Formal Methods	R	HR	SPARK : defining data flow, functional properties …
Modelling	R	HR	**Ada** and **SPARK** allow defining certain modeling properties in the code and provide means to verify them.
Structured Methodology	HR	HR	Structured Methodology designs can be implemented with **Ada**.

A.4 Software Design and Implementation

Technique/Measure	SIL 2	SIL 3/4	Comment
Formal methods	R	HR	**Ada** or **SPARK** => requirements and interface can be written in the form of formal boolean properties.
Modelling	HR	HR	**Ada** and **SPARK** => defining certain modeling properties in the code
Structured methodology	HR	HR	Ada => Structured Methodology designs can be implemented
Modular approach	M	M	Ada => A module can be represented as an **Ada** package (Interface, private part, children)
Components	HR	HR	Ada => Set of packages, project file (GPR file)
Design and coding standard	HR	M	**GNAT** compiler + **GNATcheck** + **GNAT2XML/ASIS**
Analyzable programs	HR	HR	**Ada** => improved program analysis (type ranges, parameter modes, and encapsulations) **GNATmetrics** + **GNATcheck** => monitoring code complexity **CodePeer + SPARK =>** assessment of program analyzability
Strongly typed programming language	HR	HR	**Ada**
Structured programming	HR	HR	**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	**Ada** can be used for most of the development, with potential connection to other languages such as C or assembly.

A.5 Verification and Testing

Technique/Measure	SIL 2	SIL 3/4	Comment
Formal proofs	R	HR	**Ada** pre and post conditions + **SPARK** subset => formal verification of compliance implementation / contracts.
Static analysis	HR	HR	See table A.19.
Dynamic analysis and testing	HR	HR	See table A.13.
Metrics	R	R	**GNATmetric** can retrieve metrics, such as code size, comment percentage, cyclomatic complexity, unit nesting, and loop nesting. These can then be compared with standards.
Software error effect analysis	R	HR	**GPS** support 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	See table A.21.
Functional / black-box testing	HR	HR	See table A.14.
Interface testing	HR	HR	Ada strong typing function contracts

A.6 Integration

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

A.7 Overall Software Testing

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

A.8 Software Analysis Techniques

Technique/Measure	SIL 2	SIL 3/4	Comment
Static Software Analysis	HR	HR	See table A.19.
Dynamic Software Analysis	R	HR	See table A.13 / A.14.
Software Error Effect Analysis	R	HR	**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.

A.9 Software Quality Assurance

- 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	Comment
Data Recording and Analysis	HR	M	The data produced by tools can be written to files and put in configuration management systems.

A.10 Software Maintenance

Technique/Measure	SIL 2	SIL 3/4	Comment
Impact Analysis	HR	M	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	AdaCore tools are driven from the command line and produce result files including the date and version of the tool used.

A.11 Data Preparation Techniques

Technique/Measure	SIL 2	SIL 3/4	Comment
Tabular Specification Methods	R	R	Tables of data can be expressed using the **Ada** language, together with type-wide contracts (predicates or invariants).
Formal design reviews	HR	HR	**GPS** can display code and navigate through the code as a support for walkthrough activities.
Formal proof of correctness (of data)	-	HR	When contracts on tables are expressed within the **SPARK** subset, the correctness of these contracts can be formally verified.
Walkthrough	R	HR	**GPS** can display code and navigate through the code as a support for walkthrough activities.

A.12 Coding Standards

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	Comment
Coding Standard	HR	M	**GNATcheck**
Coding Style Guide	HR	HR	**GNATcheck**
No Dynamic Objects	R	HR	**GNATcheck**
No Dynamic Variables	R	HR	**GNATcheck**
Limited Use of Pointers	R	R	**GNATcheck**
Limited Use of Recursion	R	HR	**GNATcheck**
No Unconditional Jumps	HR	HR	**GNATcheck**
Limited size and complexity of Functions, Subroutines and Methods	HR	HR	**GNATmetrics** can compute complexity **GNATcheck** can report excessive complexity.
Entry/Exit Point strategy for Functions, Subroutines and Methods	HR	HR	**GNATcheck**
Limited number of subroutine parameters	R	R	**GNATcheck**
Limited use of Global Variables	HR	M	**GNATcheck** **SPARK** can enforce documentation and verification of functions side effects, including usage of global variables.

A.13 Dynamic Analysis and Testing

Technique/Measure	SIL 2	SIL 3/4	Comment
Test Case Execution from Boundary Value Analysis	HR	HR	**GNATtest** can generate and execute a testing framework for an actual test written by developers from requirements.
Equivalence Classes and Input Partition Testing	R	HR	**Ada** and **SPARK** provide specific features for partitioning function input and verifying that this partitioning is well formed (i.e., no overlap and no hole).
Structure-Base Testing	R	HR	See table A.21.

A.14 Functional/Black Box Test

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

Technique/Measure	SIL 2	SIL 3/4	Comment
Boundary Value Analysis	R	HR	**GNATtest** can be used to implement tests coming from boundary value analysis.
Equivalence Classes and Input Partitioning Testing	R	HR	**Ada** and **SPARK** provide specific features for partitioning function input and verifying that this partitioning is well formed (i.e., no overlap and no hole).

A.15 Textual Programming Language

Technique/Measure	SIL 2	SIL 3/4	Comment
Ada	HR	HR	**GNAT Pro** tools support all versions of the Ada language.
C or C++	R	R	The **GNAT Pro** compiler supports C and C++.

A.17 Modelling

Technique/Measure	SIL 2	SIL 3/4	Comment
Data Modelling	R	HR	**Ada** allows modelling data constraints, in the form of type predicates.
Data Flow Diagram	R	HR	**SPARK** allows defining data flow dependences at subprogram specification.
Formal Methods	R	HR	**Ada** and **SPARK** allow defining formal properties on the code that can be verified by the **SPARK** toolset.

A.18 Performance Testing

Technique/Measure	SIL 2	SIL 3/4	Comment
Response Timing and Memory Constraints	HR	HR	**GNATstack** can statically analyse stack usage.

A.19 Static Analysis

Technique/Measure	SIL 2	SIL 3/4	Comment
Boundary Value Analysis	R	HR	**CodePeer** can computes boundary values for variables and parameters from the source code. **CodePeer** and **SPARK** can provide various verifications looking at potential values and boundaries values of variables such as detection of attempts to dereference a variable that could be null, values outside the bounds of an Ada type or subtype, buffer overflows, numeric overflows or wraparounds, and divisions by zero. It can also help confirming expected boundary values of variables and parameters coming from the design.
Control Flow Analysis	HR	HR	**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 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	**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).
Walkthroughs/Design Reviews	HR	HR	**GPS** can display code and navigate through the code as a support for walkthrough activities.

A.20 Components

Technique/Measure	SIL 2	SIL 3/4	Comment
Information Encapsulation	HR	HR	**Ada** provides the necessary features to separate the interface of a module from its implementation, and enforce respect of this separation.
Parameter Number Limit	R	R	**GNATcheck** can limit the number of parameters for subroutines and report violations.
Fully Defined Interface	HR	M	**Ada** offers many features to complete interface definition, including behavior specification.

A.21 Test Coverage for Code

Technique/Measure	SIL 2	SIL 3/4	Comment
Statement	HR	HR	**GNATcoverage** provides statement-level coverage capabilities.
Branch	R	HR	**GNATcoverage** provides branch-level coverage capabilities.
Compound Condition	R	HR	**GNATcoverage** provides MC/DC coverage capabilities, which can be used as an alternative to Compound Condition.

A.22 Object Oriented Software Architecture

Technique/Measure	SIL 2	SIL 3/4	Comment
Use of suitable frames, commonly used combinations of classes and design patterns	R	HR	The conventional OO design patterns can be implemented with **Ada**.
Object Oriented Detailed Design	R	HR	See table A.23.

A.23 Object Oriented Detailed Design

Technique/Measure	SIL 2	SIL 3/4	Comment
Class should have only one objective	R	HR	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	**Ada** and **SPARK** can enfore respecting the Liskov Substitution Principle, ensuring inheritance consistency.
Depth of inheritance limited by coding standards	R	HR	**GNATcheck** can limit inheritance depth.
Overriding of operations (methods) under strict control	R	HR	**Ada** can enforce explicit notation for overriding methods.
Multiple inheritance used only for interface classes	HR	HR	**Ada** only allows multiple inheritance from interfaces.

Ada

Ada - Language overview

- **Ada is a general purpose language**
  - Designed with high-integrity / safety-critical / high-security in mind
    + strong type checking
    + contract based programming
    + detecting and responding to exceptional run-time conditions

- **Ada is evolving**
  - Defined in 83, revised in 95, 2005 and 2012

- **Ada is easy to learn**
  - Simple syntax, traditional features

- **Ada is efficient**
  - It may hurt, but you can consider it to the same order of efficiency as C++ or other native languages

Ada - Features

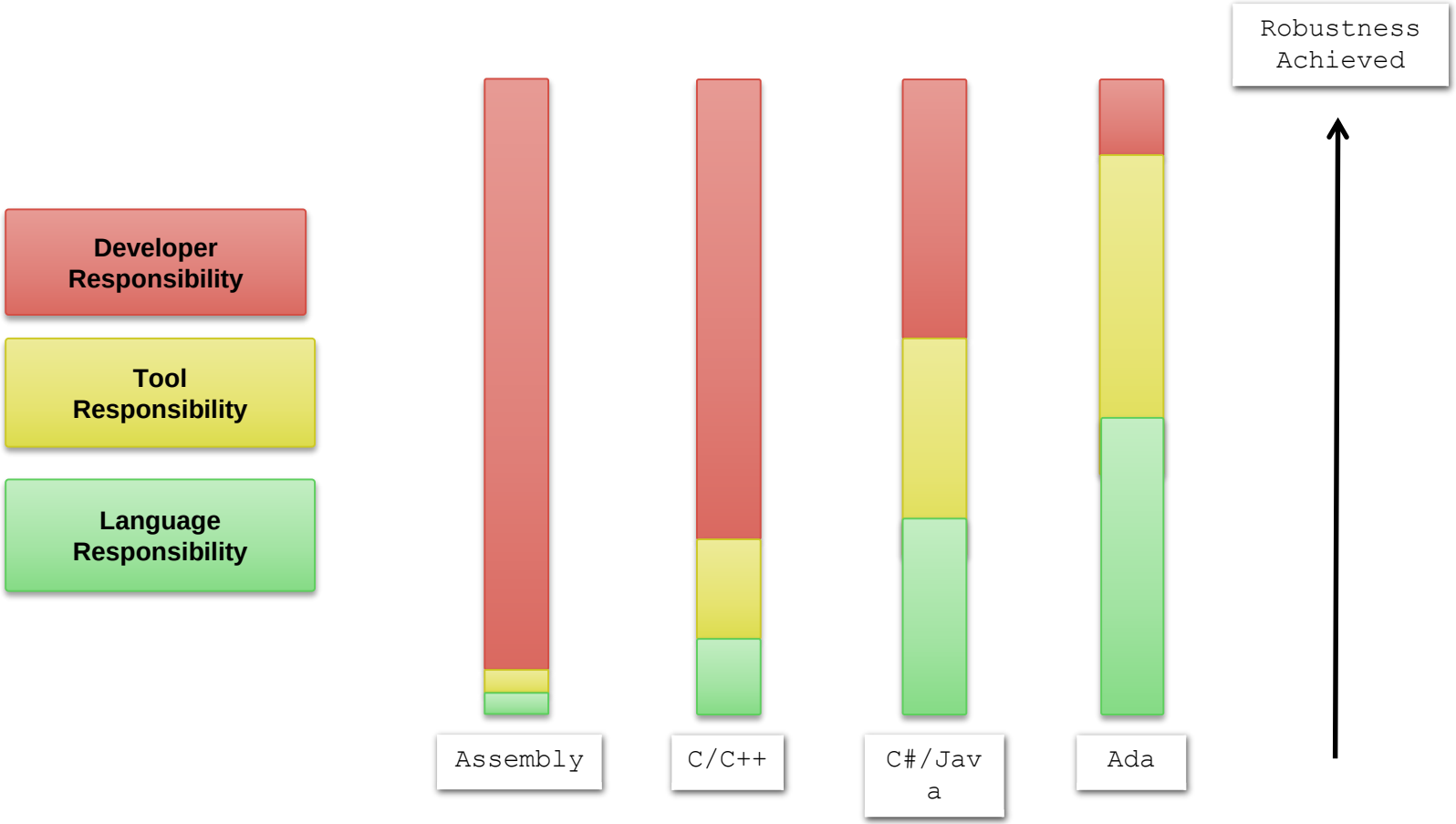
- **Ada is a general purpose language**
  - Strong typing
  - Abstractions to fit program domain
  - Generic programming/templates
  - Exception handling
  - Facilities for modular organization of code
  - Standard libraries for I/O, string handling, numeric computing, containers
  - Object-Orientated programming
  - Systems programming
  - Concurrent programming
  - Real-time programming
  - Contract based programming
  - Distributed systems programming
  - Numeric processing
  - Interfaces to other languages (C, COBOL, Fortran)
  - It contains all features of modern programming languages (encapsulation, object orientation, genericity, tasking…)

Ada - However

- **The full language is inappropriate in a safety-critical application**
  - 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). Ada is a general purpose language**

Ada benefits



Ada - conclusion

- 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

Ada – 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

SPARK

SPARK - Overview

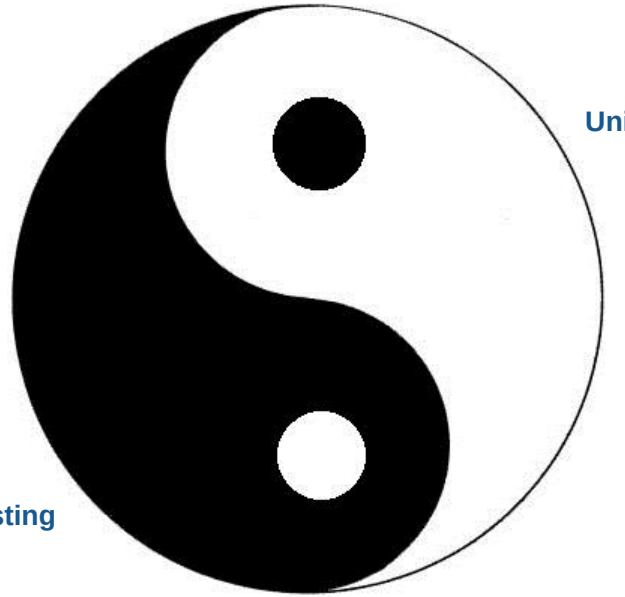
- **SPARK is a software development technology specifically designed for engineering high-reliability applications**

- **SPARK is a Ada-based language and toolset allowing to perform formal analysis**
  - Absence of run-time errors
  - Data Flow and Information Flow analysis
  - Correctness with regards to specifications / contracts

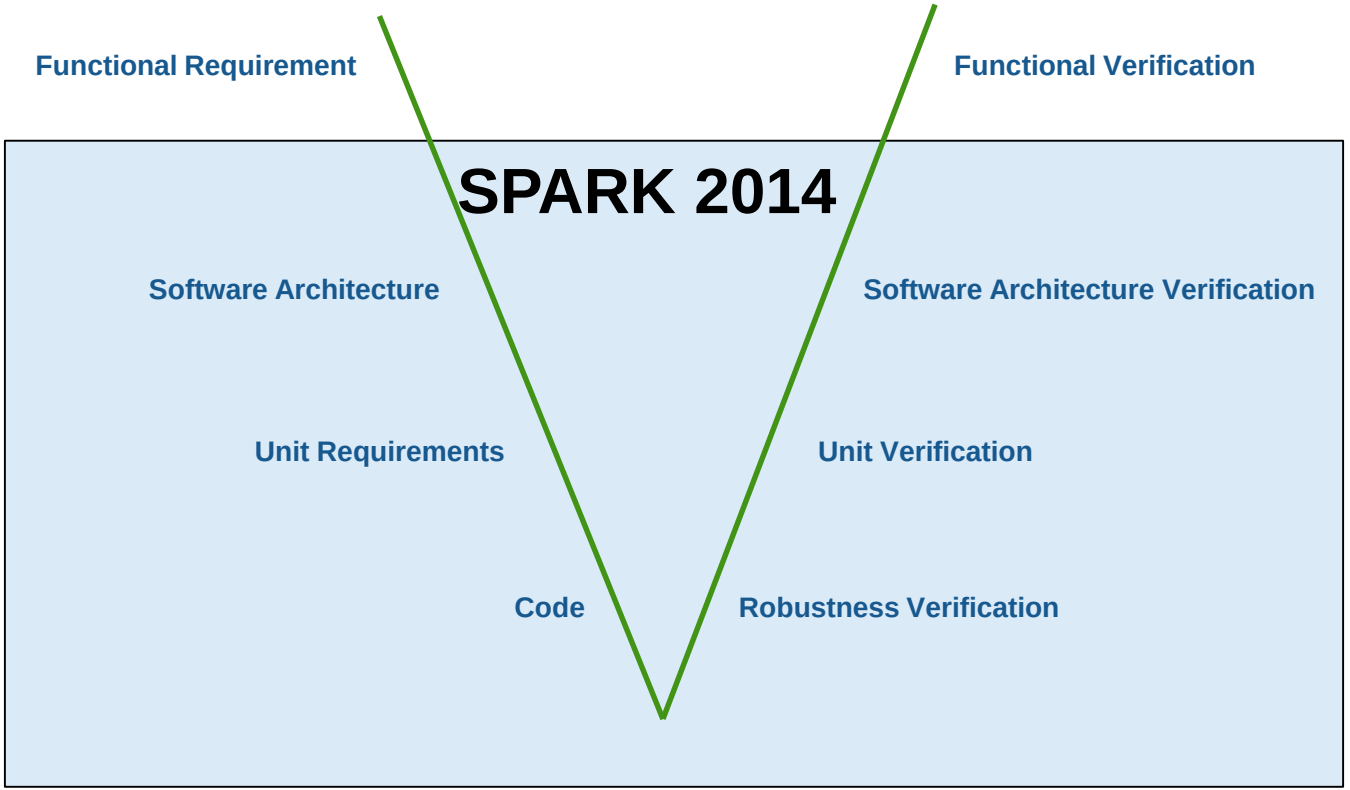
- **SPARK 2014 is a new language and toolset based on Ada 2012 including**
  - A much wider support for the Ada constructions
  - A format for contract associated with formal and executable semantics
  - An environment ready for local type substitutability proof
  - An environment ready for low-level requirement compliance and robustness verification
  - An integration environment between unit testing and unit proof

SPARK - Flexibility

SPARK 2014 code can easily be combined with full Ada code or with C



SPARK - Supporting V cycle activities



SPARK - Ada 2012 additions for contract specification

- Contracts on subprograms

```
G : Integer;

procedure P (X, Y : Integer)
   with Pre => X + Y > 0;
        Post => G = G'Old + 1;
```

- Contracts on types

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

- New expressions such as quantifiers

```
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));
```

- All of these construction are provided with dynamic semantics by Ada 2012

- All of these construction are provided with proof semantics by SPARK 2014

SPARK – Data-flow analysis

- **Data-flow analysis reveals**
  - conditional or unconditional use of undefined variables
  - unused variable definitions
  - ineffective statements

```
procedure Calc (X : in out Integer)
is
   T : Integer;
begin
   T := T + X; -- error, T has not been assigned a value
   X := T;
end Calc;
```

SPARK - Extensions to Architecture Definition

G is a global variable written
No other global variable is accessed or modified

```
G : Integer;

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

The value of G is computed from the value of X and Y

SPARK - Information-flow analysis

- **Information-flow analysis checks body against Depends annotation**

```
procedure Swap (X, Y : in out Integer)
   with Depends (X => Y,
                 Y => X);

procedure Swap (X, Y : in out Integer)
is
   T : Integer;
begin
   T := X; X := Y; Y := X;
end Swap; -- error Y doesn't depend on initial value of X
```

SPARK – Formal verification

- **Formal proof that the program satisfies specified properties**
  - RTE proof or robustness proof – verifies that no exceptions will be raised
  - Partial correctness proof – verifies the program against some specification (assuming normal termination)
  - Full correctness proof – verifies the program against some specification (including proof of normal termination)

SPARK - Annex D References

- 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

GNAT Pro toolset

GNAT Pro overview

- Configurable Run-Time Library
- Simplification of certification effort
- Traceability
- Safety-critical support and expertise

Compiler Warnings

- **Intraprocedural and low-noise (incomplete)**
- **Flow-sensitive but path-insensitive and context-insensitive**
- **Detects:**
  - Certain errors at run-time
  - Probable logic errors
  - Dead or useless code
  - Possible mismatch with user expectations
  - Representation-related mismatch
  - Portability issues

Compiler Style Checks

- **Mostly local, but can span a complete unit**
- **Style enforcement**
  - Casing, blanks, indentation, comments, line length
- **Sound engineering**
  - Maximum nesting, alphabetical order, separate specs
- **Definition of safe subset**
  - Restricted use of non-lazy and/or, explicit overriding

Restrictions

- **Define a subset of the Ada language**
- **Standard restrictions + GNAT specific**
- **Control over many features:**
  - Tasking, exceptions, dispatching, tagged types, implicit control structures, aliasing, elaboration, dependences
- **Benefits:**
  - Faster execution (different code generation)
  - Safe coding
  - Compiler and target portability

GNAT Dynamic Checks

- **Insertion of dynamic checks in binary**

- **Checks**
  - Run-time checks
  - Assertions
  - Contracts (including pre- and postconditions) [Ada2012]
  - Validity checks
  - Overflow detection
  - Uninitialized variables detection

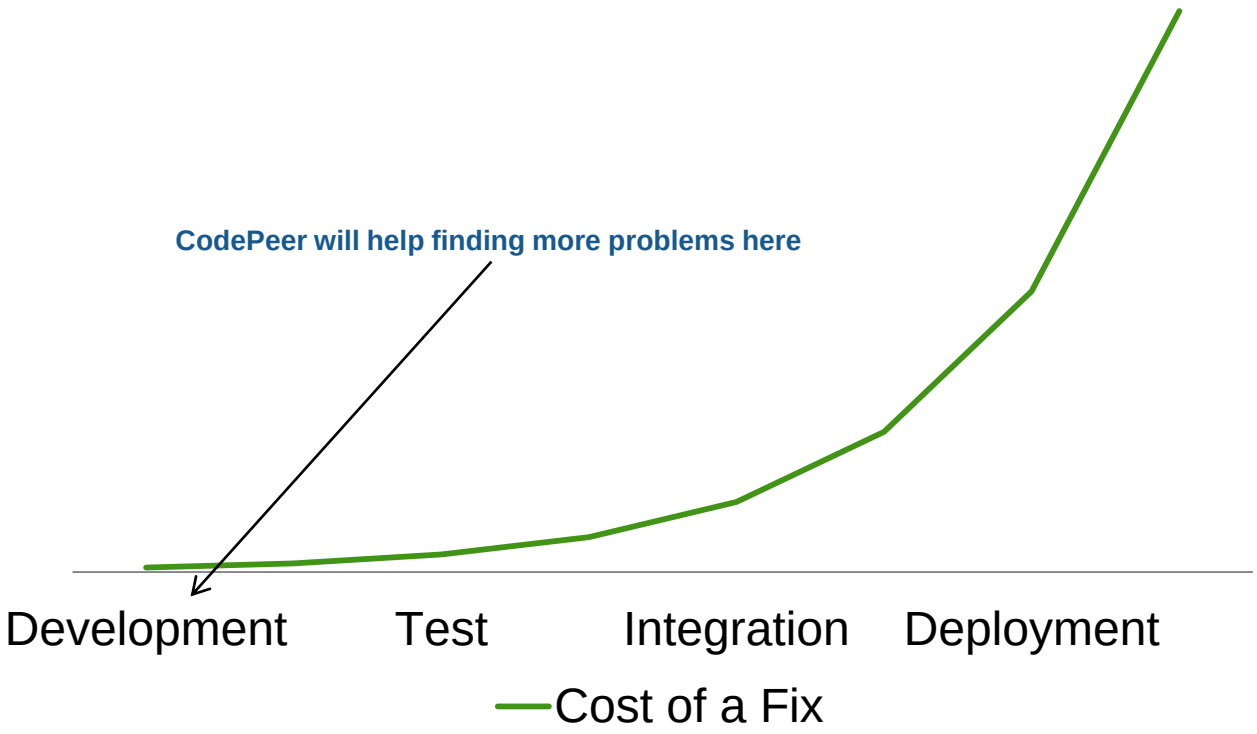
- **Controlled by:**
  - Compiler options
  - Configuration pragmas
  - Local pragmas

GNAT Pro - 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

CodePeer

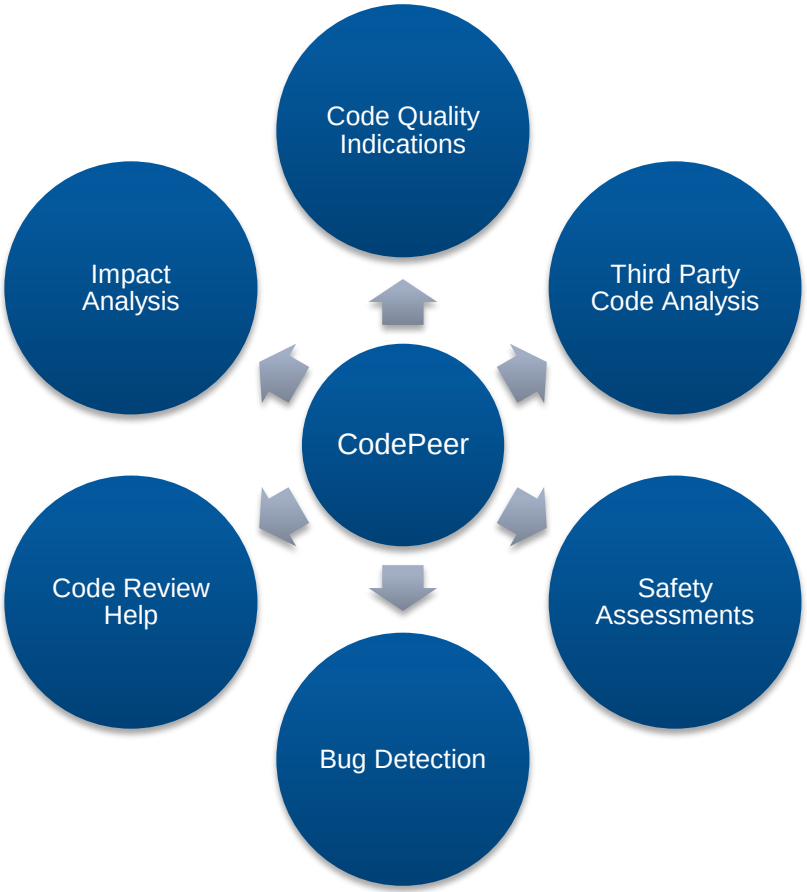
CodePeer – Reducing cost



CodePeer - Overview

- **CodePeer is an advanced static analysis tool**
  - It provides data prior to execution and test
- **It helps to identify or clear out vulnerabilities and bugs in the application**
  - Possible run-time errors
  - Useless and redundant constructs
  - Race conditions
- **It is modular and scalable**
  - Can be used on an entire project or a single file
  - Can be configured to be more or less strict, from super-warning to sound
  - Can be adapted to filter out or emphasis certain issues
  - Can concentrate on differences between baselines / versions
- **It is flexible**
  - Usable with Ada 83, 95, 2005, 2012
  - Supports most Ada compilers and targets

CodePeer - Usage

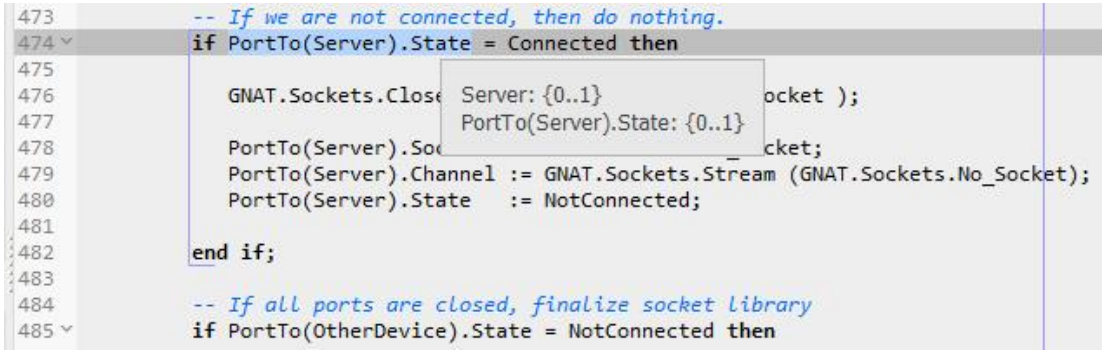


CodePeer - Errors classifications

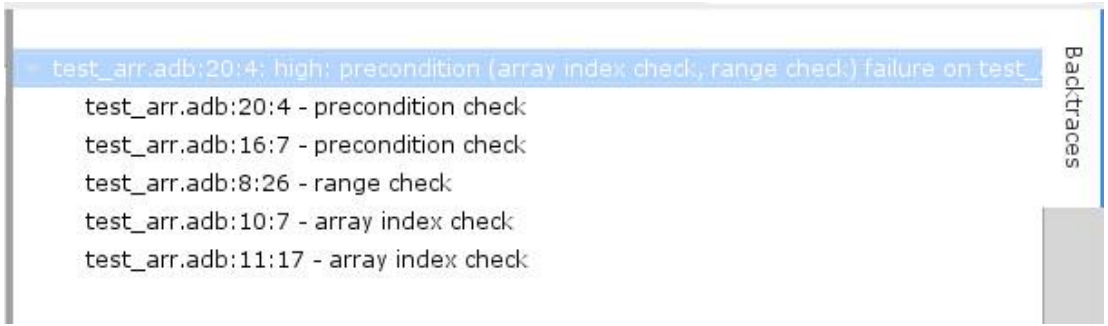
- **Run-time errors**
  - Checks added implicitly by the compiler (index checks, range checks, division by zero, …)
  - Explicit Assertions, Pre/Post Conditions, Predicates and Invariants
  - Explicitly raised exceptions
  - Uninitialized variables (both global and local)
  - Inconsistency between subprogram implicit preconditions and caller
- **Consistency/logic errors**
  - Dead code
  - Tests always true
  - Never ending loops
  - Suspicious implicit preconditions
  - Useless assignments
- **Race conditions**
  - Unprotected access to variables in multiple tasks

CodePeer - Static debugger

- **Display possible values of variables**



- **Backtrace capability on messages related to preconditions**



CodePeer - Code review

- **Messages have an associated rank**
  - Rank = severity + likelihood
  - High: certain problem
  - Medium: possible problem
  - Low: less likely problem (yet useful for completeness)
- **Filtering of messages**
  - Command line, GPS and GNATbench
  - Pattern-based automatic filtering (MessagePatterns.xml)
- **Add review status in database**
  - GPS, HTML web server
- **Add message review pragma in code**
- **Use external justification connected to output**
  - Textual output: compiler-like messages or CSV format

CodePeer - How does CodePeer work?

CodePeer computes the possible value of every variable and every expression at each program point.

It starts with leaf subprograms and propagates information up in the call-graph, iterating to handle recursion.

For each subprogram:

- It computes a precondition that guards against check failures.
- It issues check/warning messages for the subprogram.
- It computes a postcondition ensured by the subprogram.
- It uses the generated subprogram contract (precondition + postcondition) to analyze calls.

CodePeer – Annex D references

- 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

GNATcheck

GNATcheck - Coding Standard Verification

- **Automates code standard verification**
- **Can drive GNAT style checks / warnings / restrictions**
  - Extends the “simple” compiler warnings and style checks
  - Add more comprehensive rules (~ 80 rules)
- **Style-Related rules**
  - Tasking
  - Object Orientation
  - Portability
  - Program Structure
  - Programming practice
  - Readability
- **Checks various aspects of the code**
  - Feature usage
  - Metrics violation
  - Portability
  - Programming practies
- **Rules can be exempted if documented by a pragma**

GNATcheck - Examples of rules

- Limit tagged types derivation depth
- Forbid non-standard attributes
- Forbid goto statements
- Forbid use clause
- Forces explicit “in” mode
- Forbid equality between floats
- Forbid default parameters
- Forbid multiple return statement
- Force identifier naming conventions
- Forbid recursively
- Forbid unassigned out parameters
- …

GNATcheck - Annex D References

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

GNATmetrics

GNATmetrics – Metrics of Code Quality

- **Various kinds of lines counting (all, comment, blank, code…)**
- **Syntactical metrics**
  - Source Lines
  - Nesting
  - Entities (subprograms, types)
- **Complexity**
  - McCabe Cyclomatic
  - McCabe Essential
  - Loop nesting
- **Coupling (fan-out / fan-in)**

GNATmetrics - Annex D References

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

GNATdoc

GNATdoc –Adocumentation

- **Generates an HTML documentation from your source code**
- **Syntax highlighting**
- **Cross references**
- **Multiple indexes**
  - Packages, entities, source files, inheritance tree …
- **Extracts inline information from comments**

GNATtest

GNATtest - Why Automate the Process?

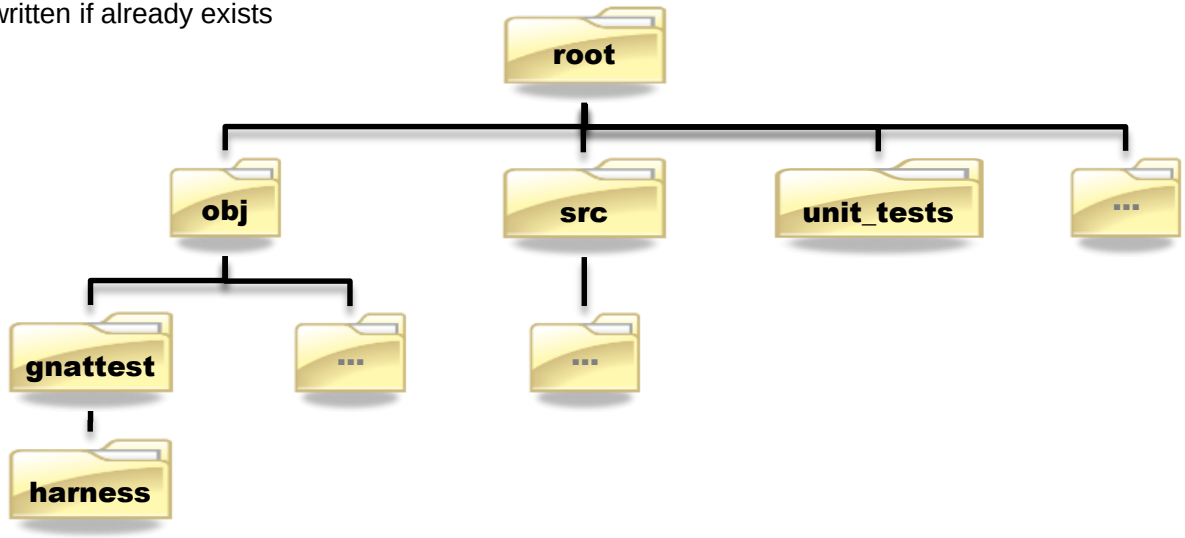
- **Developing tests is labor-intensive**
- **Much of the effort is not specific to the tests**
  - Developing the harness and driver
    - How to test generic units, etc.
  - Verifying output is as expected
  - Maintenance and update when new units to be tested
- **Ideally developers should concentrate on the high-value part: the test cases themselves**

GNATtest - Overview

- **Automatically creates and maintains harness code, and unit test skeletons for each subprogram to be tested**
- **Developers can thus focus on the high-value task of writing the actual test cases**
- **Especially valuable in systems requiring high levels of reliability, safety, and/or security**
  - Simplifies effort required to implement required test procedures
  - Can use GNATcoverage to verify test completeness
- **Supports contract-based programming tests**
- **Fully integrated into GPS**
- **Supports native, cross and high-integrity platforms**

GNATtest - Generated Outputs

- **Automatic harness code (driver infrastructure)**
  - Can be destroyed and regenerated at will
- **Unit test skeletons (actual unit test code)**
  - One for each visible subprogram in packages to be tested
  - You manually modify for specific tests' logic
  - *Not* overwritten if already exists



GNATtest - Annex D references

- **D.50 Structure Based Testing**

GNATemulator

GNATemulator - Overview

- **Fast emulation of target architecture**
  - Translates PowerPC instructions into x86 on the fly

```
$> powerpc-elf-gnatemu executable
```

- **Emulates a processor with some simple devices**
- **Additional devices can be implemented through the GNAT Bus mechanism**
- **GNATemulator is ideal to run unit testing while exercising:**
  - The target compiler
  - The appropriate data representation (endianness)

GNATemulator - Annex D references

- **D.50 Structure Based Testing**

GNATcoverage

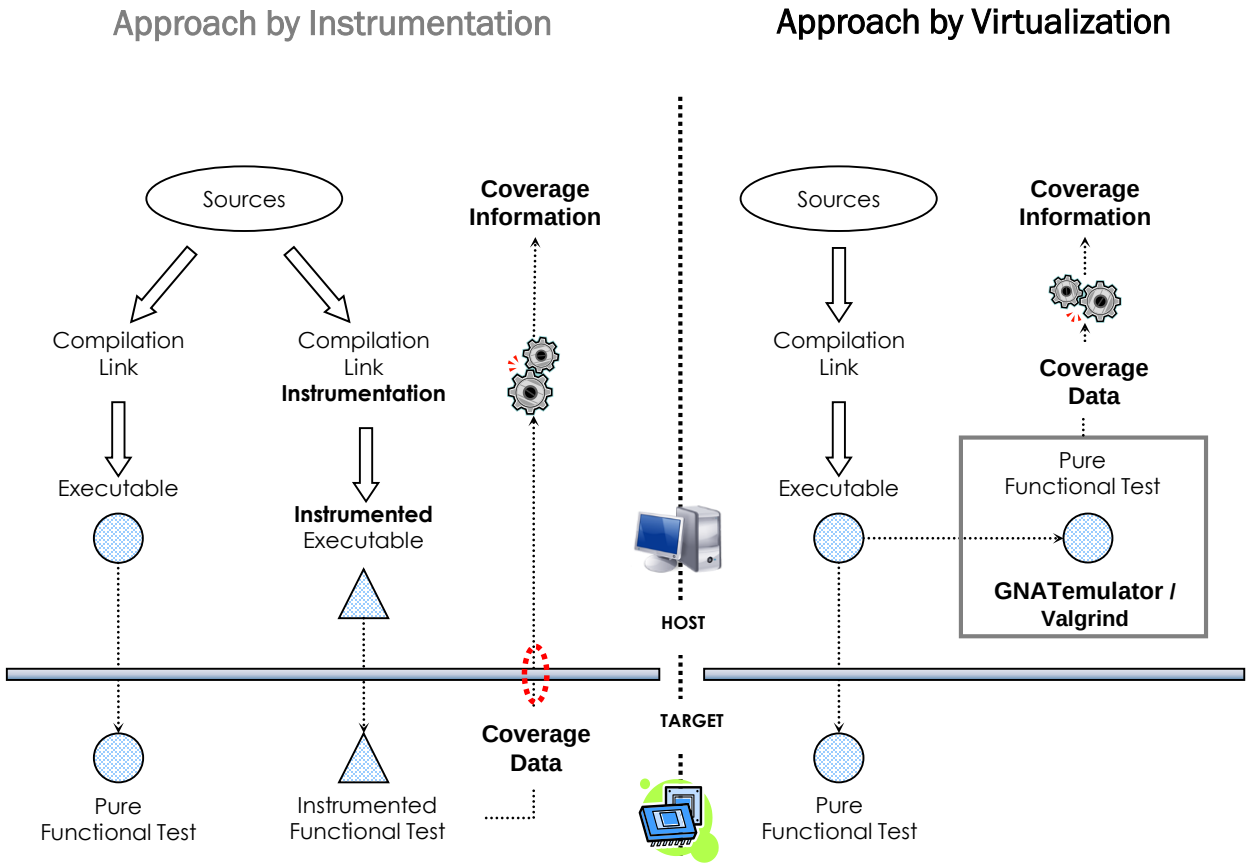
GNATcoverage - overview

- **Coverage Analysis - Multiple Modes of Operation**
  - Object coverage
    - Instruction
    - Branch
  - Source coverage
    - Statement
    - Decision
    - Modified Condition / Decision Coverage (MC/DC)
- **Run and Capture Execution Trace Data**
- **Coverage Analysis of Execution Trace Data**
- **Coverage analysis on target for boards with trace capabilities**

GNATcoverage – Build considerations

- **Compiler switches**
  - **-g**
    - Debug data
  - **-fpreserve-control-flow**
    - Control Optimizers for precise SLOC info
  - **-fdump-scos**
    - Source Coverage Obligation in *.ali files
  - **up to -O1**
    - Support for Optimizations (upto -O1)
- **Inlining Allowed (-gnatn)**
- **No External Libraries Needed**
- **All can be achieved using GNAT Project file scenario variables**

GNATcoverage - Instrumenting the Execution Platform



GNATcoverage - Output

- **xcov - Annotated Sources in Text Format**
- **report - Textual Summary**
- **HTML - Colours, Sortable Columns and Per-project indexes**

GNATcoverage report

Coverage level: stmt+decision

⇧ Show traces table

› Overview

	Total lines	Covered	Partially Covered	Not Covered	Summary
Total	18	16 89%	1 6%	1 6%	

Projects	Total lines	Covered	Partially Covered	Not Covered	Summary
Ex3	18	16 89%	1 6%	1 6%	

› Ex3

Sources	Total lines	Covered	Partially Covered	Not Covered	Summary
main.adb	8	8 100%	0 0%	0 0%	
simple.adb	10	8 80%	1 10%	1 10%	

The results (total and per file) contain:

- the total number of lines "of relevance" for the unit (definition below);
- the number of such lines that are considered as fully, partially, or not covered for the chosen coverage criteria;
- the number of such lines that are part of an exemption region, with or without actually exempted violations
- a visual summary of this coverage data.

"line of relevance" are the source lines that have associated object code and which include all or part of a source entity of interest if we are assessing a source level criterion. Source comment lines are never included in the counts, typically. In the visual summaries, the colors have the following meaning:

Covered | Partially Covered | Not Covered

GNATcoverage - Annex D references

- **D.50 Structure Based Testing**

GNATstack

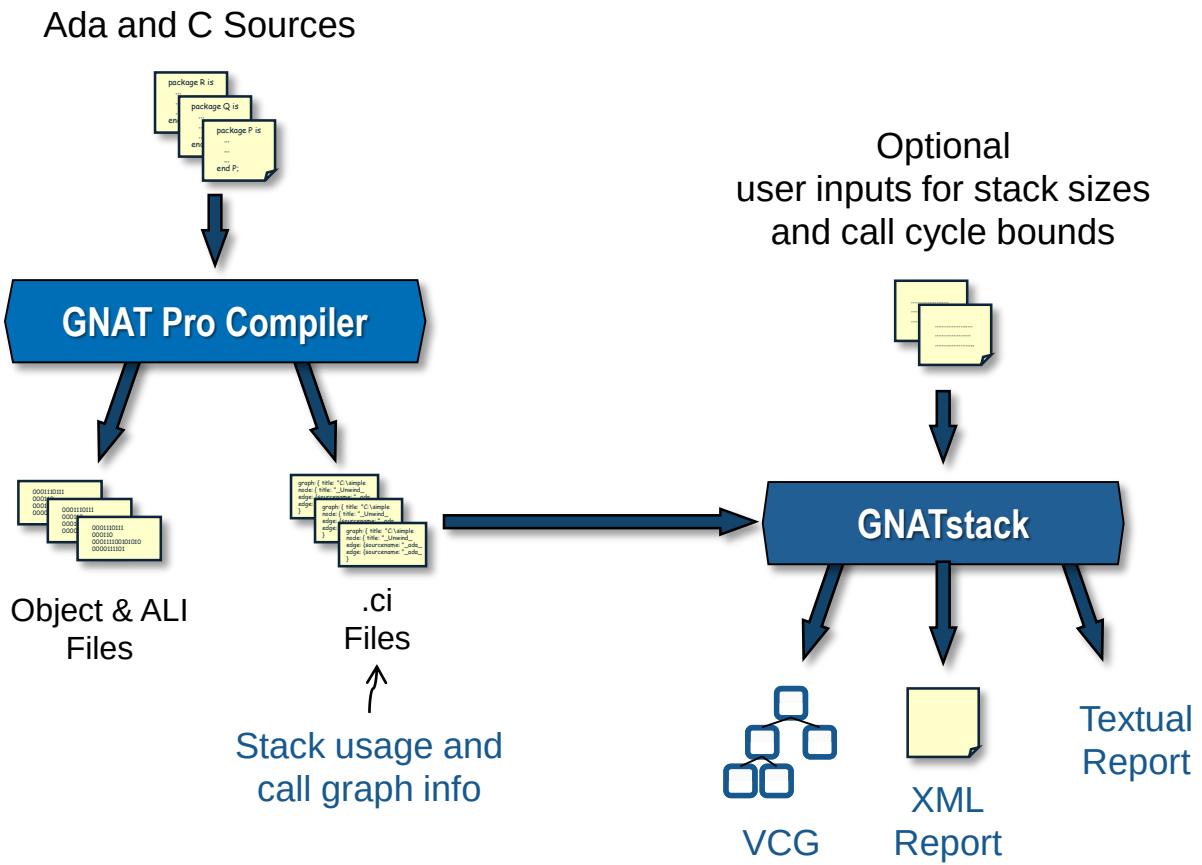
GNATstack - Overview

- **A static analyzer for maximum stack requirements**
  - Max per subprogram
  - Max per task
- **Conservative: indicates when it's not accurate**
- **Sound: believable when it says it is accurate**
- **Especially useful in a high-integrity context**
  - You must know memory usage prior to execution
  - You have access to all source code
  - Coding standards likely preclude problematic constructs

GNATstack - Capabilities

- Supports both Ada and C (and C++ “soon”)
- Supports all targets (native and cross)
- Does not involve executing the application
- Can determine actual worst case utilization
- Allows any optimization level
- Handles concurrent programs
- Handles dynamic dispatching
- Available on command-line
- Integrated with GNAT Programming Studio (GNAT Pro IDE)

GNATstack - Workflow



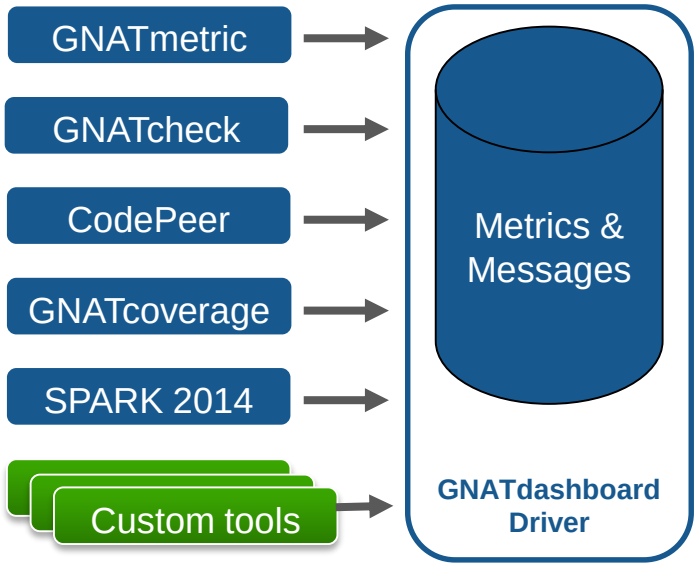
GNATdashboard

GNATdashboard - Overview

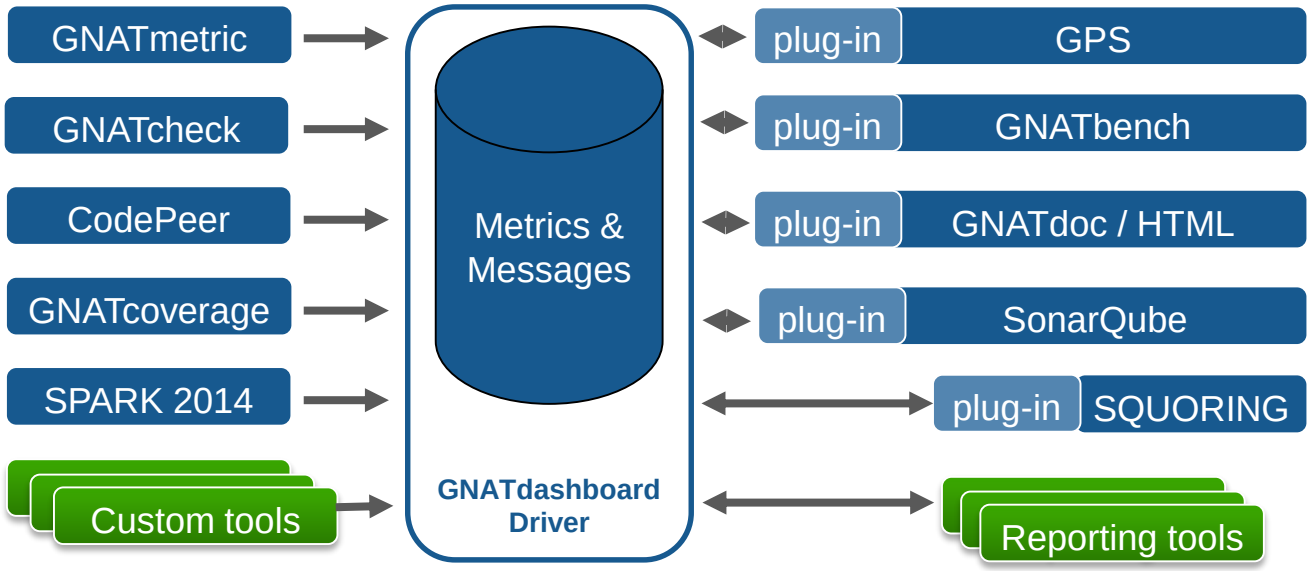
- **GNATdashboard integrates and aggregates the results of AdaCore's various static and dynamic analysis tools helping quality assurance managers and project leaders understand or reduce their software's technical debt, and eliminating the need for manual input.**
- **Provides a common interface to view code quality data and metrics such as:**
  - Code complexity
  - Code coverage
  - Conformance with standards
- **Fits into a continuous integration environment**
- **Uses project files to configure, run and analyze tool output**
- **Integrates with the SQUORE and SonarQube platforms to visualize the result**

GNATdashboard - Architecture

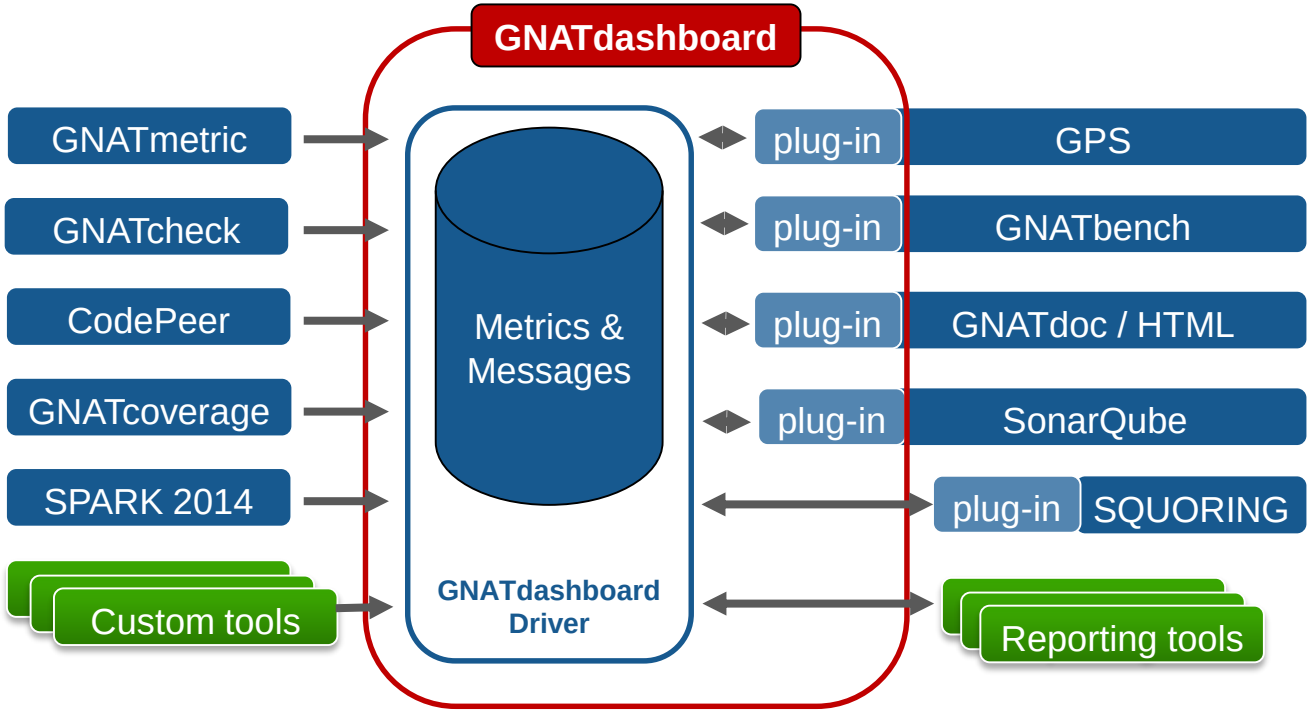
Architecture



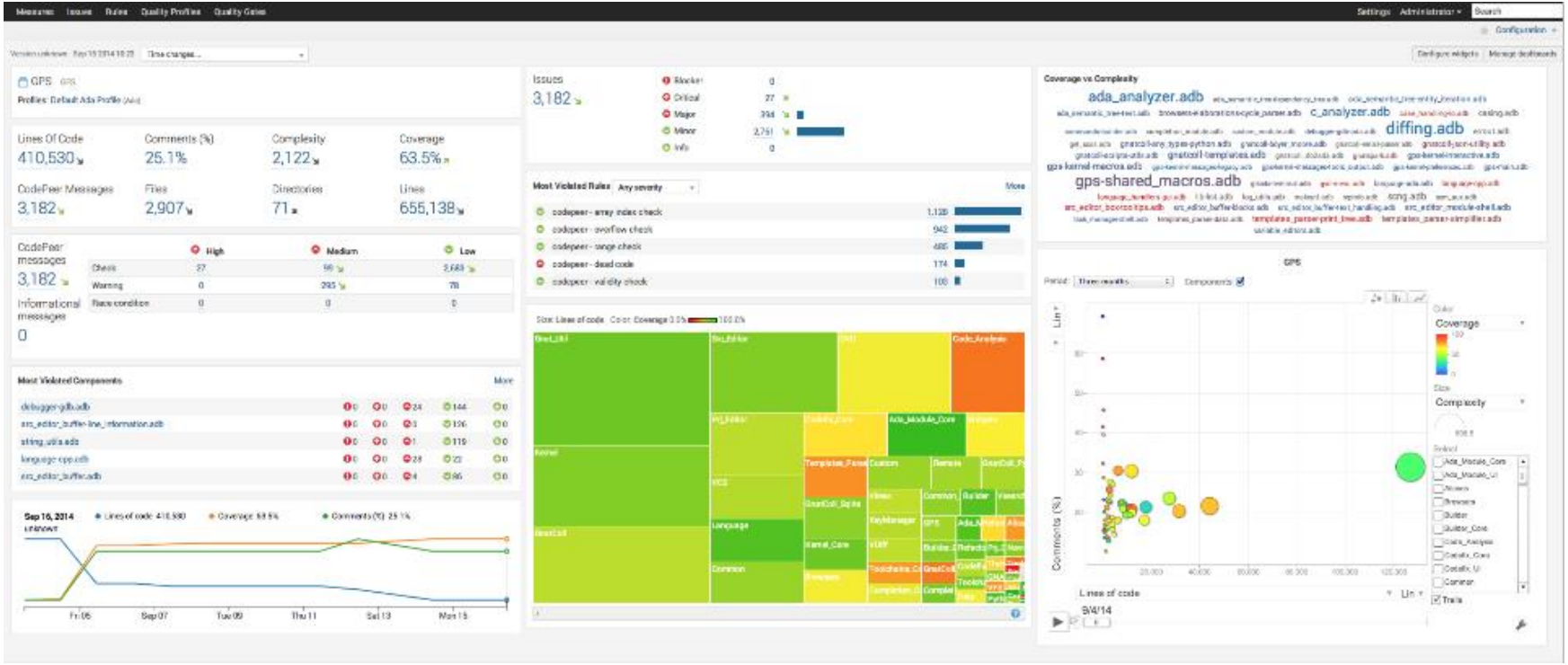
Architecture



Architecture



GNATdashboard + SonarQube



Conclusion

- AdaCore has verification tools that are applicable to the different phases in the classic V-model.
- Presented tools have been or can be qualified as EN 50128 T2 tool
- GNAT compiler has been qualified as EN 50128 T3 tool
- Developers can leverage the increase in tool responsibility to continuously verify and catch defects early.