GRAMMATECH

SIMPLIFYING DO-178B/C CERTIFICATION WITH GRAMMATECH'S CODESONAR

TRUSTED LEADERS OF SOFTWARE ASSURANCE AND ADVANCED CYBER-SECURITY SOLUTIONS

WWW.GRAMMATECH.COM

INTRODUCTION

The DO-178B (and more recently-updated DO-178C) "Software Considerations in Airborne Systems and Equipment Certification"[1] software standard was published by RTCA, Inc and developed jointly with EUROCAE, the European Organization for Civil Aviation Equipment. Designed for international use, it provides production guidelines for software used in airborne systems and equipment, which consequently must "comply with airworthiness requirements." This document describes how GrammaTech's CodeSonar® can be used to support an organization's DO-178B activities.

CodeSonar performs a whole-program, interprocedural analysis on C and C++ source code, identifying programming bugs that can result in system crashes, memory corruption, security vulnerabilities, and other serious problems. It includes numerous workflow automation features, including an API for custom integrations and support for extensions that add custom checks. CodeSonar finds bugs automatically and its unique code visualization feature makes reviewing code easier and faster.

The use of CodeSonar is most applicable to DO-178B chapters 6, 7, 11, and 12, so these are the chapters discussed in this paper, although CodeSonar can also support various activities and objectives from other chapters. In particular, CodeSonar can provide value throughout the Software Development Process (chapter 5), and the completion of a CodeSonar analysis – perhaps with a limit on the permissible number of warnings – is a useful transition criterion (chapter 4) in many cases. In places where DO-178B and DO-178C differ significantly[6], annotations are made in the text.

TYPOGRAPHICAL CONVENTIONS

The following typographical conventions are used in this document.

- CodeSonar warning class names are rendered in bold italic: ***Null Pointer Dereference***. If the warning class is disabled by default, the class name is marked with an asterisk: ***Recursive Macro****.
- Software life cycle data artifacts defined in DO-178B Chapter 11 are capitalized: Software Design Standards.
- Direct quotes from the DO-178B document are formatted like the following:
  "The rapid increase in the use of software in airborne systems and equipment used on aircraft in engines in the early 1980s resulted in a need for industry-accepted guidance for satisfying airworthiness requirements."

DO-178B CHAPTER 6: "SOFTWARE VERIFICATION PROCESS"

> *"The purpose of the software verification process is to detect and report errors that may have been introduced during the software development processes."*

CodeSonar brings powerful static analysis capabilities to the Software Verification Process. DO178-B sections 6.3 (Software Reviews and Analyses) and 6.4 (Software Testing Process) describe verification objectives that are well-supported by CodeSonar, as described below.

6.3 SOFTWARE REVIEWS AND ANALYSES

> *"Reviews and analyses are applied to the results of the software development processes and software verification process. One distinction between reviews and analyses is that analyses provide repeatable evidence of correctness and reviews provide a qualitative assessment of correctness."*

It is useful to think of CodeSonar as performing analyses whose outputs are used to inform manual *reviews*.

6.3.3 REVIEWS AND ANALYSES OF THE SOFTWARE ARCHITECTURE

CodeSonar provides support for architecture review. CodeSonar's architecture visualization, built-in checks, and extension capabilities allow many architecture problems to be flagged automatically.

> *"a. Compatibility with the high-level requirements: The objective is to ensure that the software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes."*

CodeSonar's architecture visualization feature provides interactive browsing of the various structural elements, while the optional programming API provides automation capabilities and broad potential for customization. Both GUI and API provide powerful built-in queries, such as *slicing*, that are used to check the integrity of partitioning schemes.

> *"b. Consistency: The objective is to ensure that a correct relationship exists between the components of the software architecture. This relationship exists via data flow and control flow."*

CodeSonar allows users to examine both control-flow and data-flow from a number of perspectives. Among those most useful for inspecting the relationship between software architecture components are the *control flow graph* and *call graph*. The results of control- and data-related queries can be superimposed on these graphs, providing a straightforward depiction of the dependences involved.

> *"c. Compatibility with the target computer: The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization and interrupts, between the software architecture and the hardware/software features of the target computer."*

Several built-in CodeSonar checks can detect potential compatibility problems, including ***Uninitialized Variable, Double Lock, Double Unlock, Try-lock that will never succeed***, and ***Shift Amount Exceeds Bit Width***.

> *"d. Verifiability: The objective is to ensure that the software architecture can be verified, for example, there are no unbounded recursive algorithms."*

Although demonstrating the complete absence of unbounded recursion is of course provably impossible in general, CodeSonar offers several checks that help reduce the likelihood that infinite recursion will occur. If the system architects choose to forbid recursion entirely, the ***Recursion**** and ***Recursive Macro**** checks can be enabled in order to detect violations. If recursion is to be permitted, the ***Excessive Stack Depth**** check can identify potential cases of runaway recursion. Similarly, CodeSonar's ***Potential Unbounded Loop**** check does not and cannot detect all infinite iterative loops in all programs, but will identify many loops whose boundedness cannot be established.

> *"e. Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations to the standards are justified, especially complexity restrictions and design constructs that would not comply with the system safety objectives."*

CodeSonar computes many standard complexity metrics, whose values can be compared against limits set out in the Software Design Standards. The metrics are computed at the project, file, and function levels, allowing designers to easily identify problem areas.

If system safety objectives are such that specific design constructs must be restricted or forbidden, custom CodeSonar checks can be written to issue warnings whenever those constructs are encountered.

> *"f. Partitioning integrity: The objective is to ensure that partitioning breaches are prevented or isolated."*

CodeSonar allows users to trace both control and data through the various possible executions of a program. Users can obtain answers to such questions as "can data in region A influence execution in region B," "is there an execution path between region C and region D," and "what are the callers of function F?"

Custom CodeSonar checks can also be used to detect partitioning breaches automatically.

Explicit control and data flow may not be the only avenues for breaching partitioning requirements. In some cases other channels, such as the file system, must be considered. Designers may elect to forbid use of the file system entirely, in which case custom CodeSonar checks for uses of file-related functions and data types will be extremely useful. Alternatively, file usage may be permitted under restricted circumstances, in which case the custom checks will need to verify that the appropriate conditions hold whenever files are used.

6.3.4 REVIEWS AND ANALYSES OF THE SOURCE CODE

Source code analysis is CodeSonar's specialty. A full account of the reviews and analyses supported by these products is not possible in this space; instead we outline some key features.

> *"b. Compliance with the software architecture: The objective is to ensure that the Source Code matches the data flow and control flow defined in the software architecture."*

CodeSonar provides powerful functionality for visualizing data and control flow in software, and for answering complex queries about both. Users can compare the flow visualized to that defined in the software architecture to identify any discrepancies.

If control flow requirements specify a particular operation sequence that must be adhered to, CodeSonar can be extended with a custom check that issues warnings whenever the operations appear out of order.

> *"c. Verifiability. The objective is to ensure that the Source Code does not contain statements and structures that cannot be verified and that the code does not have to be altered to test it."*

CodeSonar observes the normal build process for a software project and uses the information thus gained to construct an internal representation that is then subjected to various analyses. The production code does not need to be altered in any way.

"d. Conformance to standards: The objective is to ensure that the Software Code Standards were followed during the development of the code, especially complexity restrictions and code constraints that would be consistent with the system safety objectives...."

CodeSonar provides significant support for upholding coding standards, both through built-in checks for standard rule sets such as Power of Ten[4] and through a rich API that allows users to implement custom checks in support of local code standards. **11.7 Software Design Standards** and **11.8 Software Code Standards** detail the ways in which CodeSonar can support and enforce a wide range of software standards.

"f. Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including stack usage, fixed point arithmetic overflow, resource contention, worst-case execution timing, exception handling, use of uninitialized variables or constants, unused variables or constants, and data corruption due to task or interrupt conflicts."

CodeSonar performs a large number of correctness and consistency checks, including those for the following warning classes:

- ***Excessive Stack Depth****
- ***Buffer Overrun, Buffer Underrun***
- ***Type Overrun, Type Underrun***
- ***Cast Alters Value, Integer Overflow of Allocation Size***
- ***Uninitialized Variable***
- ***Unused Value***
- ***Deadlock, File System Race Condition***

DO-178C adds guidance on evaluating the compiler/linker tool chain output and execution.

"The compiler (including its options), the linker (including its options), and some hardware features may have an impact on the worst-case execution timing and this impact should be assessed. Also added floating-point arithmetic as a consideration."

CodeSonar's unique binary analysis capability detects defects and security vulnerabilities in compiled object code and executables, allowing developers to assess the quality of the compiled code as well as all third-party code compiled into the product.

6.4 SOFTWARE TESTING PROCESS

CodeSonar's static analysis can detect several of the problems listed as identifiable through requirements-based testing methods, and can help resolve questions arising from test-coverage analysis.

6.4.3 REQUIREMENTS-BASED TESTING METHODS

Static analysis is not a substitute for testing but a complement to it, and in some cases will find problems that tests miss.

"a. Requirements-Based Hardware/Software Integration Testing"
"b. Requirements-Based Software Integration Testing"
"c. Requirements-Based Low-Level Testing"

CodeSonar detects many of the "typical errors revealed by [these] testing method[s]" early in the software development process. The following table shows relevant CodeSonar warning classes:

ERROR	CODESONAR WARNING CLASS(ES)
"Stack overflow."	***Excessive Stack Depth****
"Incorrect initialization of variables and constraints."	***Uninitialized Variable, Double Initialization***
"Parameter passing errors."	***Function Call Has No Effect, Unreasonable Size Argument***
"Data corruption, especially global data."	***Buffer Overrun, Buffer Underrun, Type Overrun, Type Underrun, MAX_PATH Exceeded, No Space for Null Terminator, Return Pointer to Freed, Return Pointer to Local, Use After Close, Use After Free***
"Inadequate end-to-end numerical resolution."	***Cast Alters Value***
"Incorrect sequencing of events and operations."	***Double Lock, Double Unlock, Try-lock that will never succeed, Free Null Pointer, Null Test After Dereference***
"Incorrect loop operations"	***Potential Unbounded Loop****

6.4.4 TEST COVERAGE ANALYSIS

> *"Test coverage analysis is a two step process, involving requirements-based coverage analysis and structural coverage analysis. The first step analyzes the test cases in relation to the software requirements to confirm that the selected test cases satisfy the specified criteria. The second step confirms that the requirements-based test procedures exercised the code structure."*

CodeSonar is useful in the resolution phase of test coverage analysis, if this is required.

6.4.4.3 STRUCTURAL COVERAGE ANALYSIS RESOLUTION

> *Structural coverage analysis may reveal code structure that was not exercised during testing. Resolution would require additional software verification process activity. This unexecuted code may be the result of:*
>
> *"c. Dead code"*
> *"d. Deactivated code"*

DO-178C uses the term "extraneous" code, of which dead code and deactivated code are subsets. CodeSonar capabilities are still applicable.

In some cases, a test suite may be unable to exercise a particular code region because the region is in fact unreachable. If structural coverage analysis resolution is to include a diagnosis of dead or deactivated code, it's useful to support this diagnosis with static analysis results.

CodeSonar detects unreachable code as part of its standard analysis suite. Distinct warning classes ***Unreachable Call, Unreachable Computation, Unreachable Conditional, Unreachable Control Flow,*** and ***Unreachable Data Flow*** identify the most important element present in each unreachable region, providing the user with additional information about the consequences of not executing the code.

In some cases, code is unreachable because it is guarded by a conditional statement that can only ever evaluate one way. CodeSonar's ***Redundant Condition*** warning class is designed to identify such situations.

CodeSonar provides complementary information about code reachability at several levels of detail. Users can determine whether there are entire functions that are never called using the program *call graph*. At the statement level, a *point-mode forward slice* from the program entry point will compute all code that can be reached; any code not in this set is therefore unreachable.

Note that CodeSonar works by analyzing a specific build of a software project. To establish that code is unreachable under all applicable build configurations (for example, unreachable on all target platforms, or under all permitted preprocessor settings), the analyses must be applied to all these builds and the results compared.

DO-178B CHAPTER 7: SOFTWARE CONFIGURATION MANAGEMENT PROCESS

While CodeSonar is not a software configuration management tool, it can interact with the Software configuration management process in several useful ways.

7.2 SOFTWARE CONFIGURATION MANAGEMENT PROCESS ACTIVITIES

CodeSonar can be used in support of the following software configuration management process activities.

7.2.2 BASELINES AND TRACEABILITY

Where unambiguous labels have been assigned to configuration items, CodeSonar provides several powerful mechanisms for applying corresponding labels to analysis artifacts generated for those items.

When a labeled configuration item is analyzed with CodeSonar:

- The analysis can be *annotated* with the same label, either directly or retroactively.
- A user can manually annotate all warnings issued by the analyses with the label (all at the same time, if desired).
- A custom *warning processor* can automatically annotate each warning issued by the analysis with the label. With further customization, annotation can be restricted to a particular subset of the warnings, for example only those that were issued for the first time (that is, not seen in any previous analysis).

The flexible search in CodeSonar includes functionality that allows users to find all warnings that have a particular annotation, or that were issued by an analysis with a particular annotation. The annotation capability thus provides direct traceability between a particular configuration item and the analysis results for that item.

From time to time it may be necessary to analyze projects in which different components have different configuration identifiers. CodeSonar can provide traceability even in this case: the only differences are that the analysis annotation will need to include all relevant labels, and the warning processor (or person) responsible for annotating warnings will need to take each warning's location into account when assigning its label.

Similarly, when a baseline is established for a configuration item, CodeSonar annotations are used to identify analysis artifacts associated with that baseline.

7.2.3 PROBLEM REPORTING, TRACKING, AND CORRECTIVE ACTION

Problem reporting and tracking are key roles of CodeSonar. Each analysis of a project issues a number of warnings (possibly zero). As a project undergoes a cycle of modification and re-analysis over time, the CodeSonar hub builds up a historical record. Information is kept about each analysis, including the warnings issued, files analyzed, build settings on which the project was based, and annotations made. At a finer-grained level, the "same" warning can be tracked over time, determining the analysis in which it first appeared, and (if the underlying problem is successfully fixed) the analysis in which it first went away. CodeSonar keeps all annotations for each warning (except for cases where an analysis has been explicitly deleted), so a history of notes, priority determinations, owners, and assessments is built up over time.

The CodeSonar hub is not restricted to warnings issued by the CodeSonar analysis. For example, CodeSonar ships with an example script that uploads the warnings issued by a FindBugs™[5] analysis to a CodeSonar hub; scripts that upload analysis results from other tools with different output formats are similarly possible.

Conversely, warnings issued by the CodeSonar analysis are not restricted to the CodeSonar hub. Custom warning processors can send analysis results to external tools, and can be applied either at analysis time or retroactively to individual warnings and groups of warnings. CodeSonar ships with a processor that creates a bug report concerning one or more CodeSonar warnings and submits it to a Bugzilla[2] database.

7.2.4 CHANGE CONTROL

> *"e. Throughout the change activity, software life cycle data affected by the change should be updated and records should be maintained for the change control activity."*

As described in **7.2.3 Problem Reporting, Tracking, and Corrective Action CodeSonar** by default keeps substantial records of past and present analysis results and the relationships between them. In cases where these records must be expanded upon, they can readily be exported in a variety of formats. When traceability between a particular change and the analysis of the changed software is important, CodeSonar provides several mechanisms for making the connection explicit; these are described in **7.2.2 Baselines and Traceability**.

7.2.5 CHANGE REVIEW

Just as CodeSonar warning processors are used to apply annotations to warnings (**7.2.2 Baselines and Traceability**) or to submit them to third-party trackers (**7.2.3 Problem Reporting, Tracking, and Corrective Action**), CodeSonar can be used to assign warnings to specified reviewers. During the early development stages, a warning processor might be configured to assign each incoming warning to the engineer who last modified the source file containing the warning. Later, a warning processor might use the software level of the affected code to assign a priority to each incoming warning and then assign it to an engineer of appropriate seniority.

DO-178B CHAPTER 11: SOFTWARE LIFE CYCLE DATA

> *"Data is produced during the software life cycle to plan, direct, explain, define, record, or provide evidence of activities."*

The Software Life Cycle Data defined by DO-178B include Plans with activities that are accomplished using CodeSonar. In addition, several of the specified Standards are supported or partially supported by CodeSonar.

11.7 SOFTWARE DESIGN STANDARDS

CodeSonar is used to support and enforce a number of the Software Design Standards listed in DO-178B section 11.7.

> *"b. Naming conventions to be used."*

As described in *11.8 Software Code Standards*, CodeSonar users can author custom checks for violations of naming conventions.

> *"c. Conditions imposed on permitted design methods, for example, scheduling, and the use of interrupts and event-driven architectures, dynamic tasking, re-entry, global data, and exception handling, and rationale for their use."*

It is straightforward to write custom CodeSonar checks on the use of global data or exception handling: such checks would inspect the internal representation and issue warnings whenever the relevant artifacts were observed (or observed outside permitted contexts).

> *"e. Constraints on design, for example, exclusion of recursion, dynamic objects, data aliases, and compacted expressions."*

CodeSonar ships with several checks that support design constraints, and can be extended to add others. The ***Recursion**** and ***Recursive Macro**** warning classes behave as one might expect: CodeSonar issues warnings whenever functions (or macros, respectively) are directly or indirectly recursive. Classes ***Dynamic Allocation After Initialization**** and ***Pointer Type Inside Typedef**** are useful if the design constraints limit but do not fully exclude the use of dynamic objects. If dynamic objects are to be fully excluded, users can construct custom CodeSonar checks that issue warnings whenever allocators are used. Similarly, the CodeSonar extension mechanisms allow users to write checks for violations of restrictions on data aliasing and on expression side-effects.

> *"f. Complexity restrictions, for example, maximum level of nested calls or conditional structures, use of unconditional branches, and number of entry/exit points of code components."*

As with the design constraints, CodeSonar has built-in checks in support of some forms of complexity restriction and can be extended to add checks for others. ***Excessive Stack Depth**** warnings are issued when the function call stack exceeds a specified size: this expands on the notion of restricting call nesting by taking into account the size of the execution record for each call. Several built-in CodeSonar checks cover various forms of unconditional branch: ***Empty if Statement, Empty switch Statement, Redundant Condition, Goto Statement****, and ***Use of longjmp****. Users can create custom CodeSonar checks for violations of other complexity restrictions that might be included in a project's Software Design Standards.

11.8 SOFTWARE CODE STANDARDS

CodeSonar provides significant support for upholding coding standards, both through built-in checks for standard rule sets such as Power of Ten[4] and through a rich API that allows users to implement custom checks in support of local code standards.

> *"a. Programming language(s) to be used and/or defined subset(s)."*

The CodeSonar API provides access to the abstract syntax trees (ASTs) generated for an analyzed software project. Users can leverage this access to author custom checks that report errors when forbidden program features are used.

"c. Naming conventions for components, subprograms, variables, and constants."

Because the CodeSonar API provides full programmatic access to the internal representation generated for its analysis, users can readily create custom checks for naming convention violations and have these checks carried out as part of the CodeSonar analysis. Checks can be targeted to specific program entities so they can easily handle naming conventions that impose, for example, one set of rules governing the permissible names for variables and another for functions.

"d. Conditions and constraints imposed on permitted coding convention, such as the degree of coupling between software components and the complexity of logical or numerical expressions and rationale for their use."

Various code complexity metrics have become standard in software engineering practice. CodeSonar computes and reports a number of these standard metrics, including the McCabe[6], Halstead[3], and user-defined metrics.

The AST access provided by the CodeSonar API is useful here, as in previous cases. Given an expression complexity limit specified by the Software Code Standards, a user can create a custom check that issues a warning whenever the analysis encounters an expression whose complexity exceeds this limit.

DO-178B CHAPTER 12: ADDITIONAL CONSIDERATIONS

Of the additional considerations discussed in DO-178B Chapter 12, CodeSonar is particularly applicable to section 12.1 (Use of Previously Developed Software). The understanding, use, and modification of existing software are fundamental parts of any software development effort, and CodeSonar provides advanced capabilities for accomplishing such tasks.

12.1 USE OF PREVIOUSLY DEVELOPED SOFTWARE

"The guidelines of this subsection discuss the issues associated with the use of previously developed software..."

Two significant issues for users of previously-developed software are understanding the software and trusting that it does not contain bugs and vulnerabilities. CodeSonar aids in understanding of previously-developed software and its interaction with the overall system at multiple levels, including data and control flow, call graphs, and effects on global values. To help with establishing trust, CodeSonar is applied to previously-developed source code just as easily as it is applied to code under current development. Problems, including security vulnerabilities such as buffer overflows – whether introduced accidentally or deliberately – can thus be identified and marked for elimination. CodeSonar's binary analysis scans object code and executables for defects in code that you don't have source for, including system libraries, previously-developed software, and third-party binary code.

12.1.1 MODIFICATIONS TO PREVIOUSLY DEVELOPED SOFTWARE

"This guidance discusses modifications to previously developed software where the outputs of the previous software life cycle processes comply with this document. Modification may result from requirement changes, the detection of errors, and/or software enhancements."

CodeSonar is designed to support the iterative nature of real-world software development.

"e. Areas affected by the change should be reverified considering the guidelines of section 6."

CodeSonar supports reverification of areas affected by changes in several important ways. First, it's used to analyze partial programs. Once the areas affected by a change have been identified, those in charge of verification can thus choose to concentrate their resources on reverifying only those affected areas. Second, CodeSonar supports incremental analysis, in which only those parts of the internal representation affected by changes in the code base are rebuilt and reanalyzed. This can offer substantial time-savings when analyzing large projects. Third, the CodeSonar hub database provides a historical record of the analyses for a software project and the warnings issued by the analyses. Given a particular warning, CodeSonar users can identify the analysis that first issued the "same" warning (or a closely related one), and, if applicable, the analysis at which the warning stopped being issued (because the underlying problem was fixed).

12.1.3 CHANGE OF APPLICATION OR DEVELOPMENT ENVIRONMENT

> *"Use and modification of previously developed software may involve a new development environment, a new target processor or other hardware, or integration with other software than that used for the original application."*

The internal representations constructed by CodeSonar take into account many aspects of the software being analyzed and the build process used to construct that software, including the compiler or compilers, compiler options, preprocessor settings, and the platform for which the software is being built. The analyses carried out on these internal representations will likewise reflect these factors. When a change in application or development environment leads to a change in the project and its representation, CodeSonar can be used to analyze the scope and impact of these changes in the same ways described in *12.1.1 Modifications to Previously Developed Software*.

12.1.4 UPGRADING A DEVELOPMENT BASELINE

> *"Guidelines follow for software whose software life cycle data from a previous application are determined to be inadequate or do not satisfy the objectives of this document, due to the safety objectives associated with a new application... Reverse engineering may be used to regenerate software life cycle data that is inadequate or missing in satisfying the objectives of this document."*

Just as CodeSonar is used in support of the requirements of DO-178B Chapter 6 for software currently under development, it is also applied to previously-developed software to bring software life cycle data up to the level required.

CONCLUSION

CodeSonar can make a significant contribution to DO-178B and DO-178C activities, with its sophisticated analyses that provide multiple points of leverage for verification and reverification. CodeSonar's extensive reporting and record-keeping mechanisms also provide support for various DO-178B and DO-178C objectives.

REFERENCES

1. DO-178B, Software Considerations in Airborne Systems and Equipment Certification. 1992, RTCA Inc.

2. Halstead,M.H., Elements of Software Science. 1977, New York, NY: Elsevier. 142.

3. Holzmann,G.J., The Power of 10: Rules for Developing Safety-Critical Code. IEEE Computer, 2006. 39(6): pp. 95-97.

4. Pugh,W., FindBugs - Find Bugs in Java Programs, http://findbugs.sourceforge.net/.

5. Watson,A.H. and McCabe,T.J., Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. 1996, Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD. NIST Special Publication 500-235.

6. DO-178B/C Differences Tool, FAA, September 16, 2013.

GrammaTech, Inc. is a leading developer of software-assurance tools and advanced cyber-security solutions. GrammaTech helps organizations develop and release high quality software, free of harmful defects that cause system failures, enable data breaches, and increase corporate liabilities in today's connected world. GrammaTech's CodeSonar is used by embedded developers worldwide.