GRAMMATECH

POWER OF TEN RULES MAPPED TO CODESONAR® 6.2 WARNING CLASSES

TRUSTED LEADERS OF SOFTWARE ASSURANCE AND ADVANCED CYBER-SECURITY SOLUTIONS

WWW.GRAMMATECH.COM

INTRODUCTION

The “Power of Ten” refers to a set of ten rules developed by Gerard Holzmann of the NASA Jet Pro-pulsion Laboratory for use in writing safety-critical software. The rules are simple, but they specify strict limits on the forms code can take. These limits support code clarity and analyzability, which are especially important for safety-critical applications.

CodeSonar 6.2 includes warning classes that support checking for the Power of Ten rules. Every CodeSonar warning report includes the numbers of any Power of Ten rule that are closely mappedto the warning’s class. (The close mapping for a warning class is the set of categories—including Power of Ten rule numbers—that most closely match the class, if any).

You can configure CodeSonar to enable and disable warning classes mapped to specific Power ofTen rule, or use build presets to enable all warning classes that are closely mapped to any Powerof Ten rule. In addition, you can use the CodeSonar search function to find warnings related to a specific Power of Ten rule, or to any Power of Ten rule.

For more information on the Power of Ten

rules:http://www.spinroot.com/p10/

The following table lists the CodeSonar warning classes that are closely mapped to Power of Tenrules.

Note-All CodeSonar Power of Ten mappings are close.

GrammaTech is a leading global provider of application testing (AST) solutions used by the world’smost security conscious organizations to detect, measure, analyze and resolve vulnerabilitiesfor software they develop or use. The company is also a trusted cybersecurity and artificial intelligence research partner for the nation’s civil, defense, and intelligence agencies.

Power of 10 Rule	Closely Mapped CodeSonar C/C++ Warning Classes
1. Restrict to simple control flow constructs.	Goto Statement Recursion Use of <setjmp.h> Use of longjmp Use of setjmp Initialization Cycle Unordered Initialization
2. Give all loops a fixed upper-bound.	Potential Unbounded Loop
3. Do not use dynamic memory allocation after initialization.	Dynamic Allocation After Initialization
4. Limit functions to no more than 60 lines of text.	Function Too Long
5. Use minimally two assertions per function on average.	Not Enough Assertions
6. Declare data objects at the smallest possible level of scope.	Scope Could Be File Static Scope Could Be Local Static
7. Check the return value of non-void functions, and check the validityof function parameters.	Ignored Return Value Unchecked Parameter Dereference
8. Limit the use of the preprocessor to file inclusion and simple macros.	## Follows # Operator Conditional Compilation Macro Does Not End With } or) Macro Does Not Start With { or (Macro Name is C Keyword Macro Uses # Operator Macro Uses ## Operator Non-Boolean Preprocessor Expression Preprocessing Directives in Macro ArgumentRecursive Macro Unbalanced Parenthesis Use of <stdio.h> Input/Output Macro Use of <wchar.h> Input/Output MacroVariadic Macro
9. Limit the use of pointers. Use no more than two levels of dereferencing per expression.	Function Pointer Macro Uses [] Operator Macro Uses -> Operator Macro Uses Unary * OperatorPointer Type Inside Typedef Too Many Dereferences
10. Compile with all warnings enabled, and use one or more sourcecode analyzers.	Not All Warnings Are Enabled Warnings Not Treated As Errors