Simplifying ISO 26262 Compliance with GrammaTech Static Analysis Tools

Introduction

The modern automobile is increasingly a software-based artifact. Features and functions from climate control to braking are partially or completely controlled by embedded software. Manfred Broy of the Institut für Informatik, Technische Universität München provides some illustrative figures[4]: a premium car currently contains "more than ten million lines" of code, covering "[m]ore than 2000" functions; software and electronics together account for "[u]p to 40% of the production costs of a car".

The increased use of software yields features that can greatly enhance safety for drivers and passengers. Conversely, when safety-critical automotive functions are controlled by software, bugs can endanger lives. That the headlines are not full of terrible stories of software related car crashes is a testament to the quality of automotive software engineering. Nonetheless, problems do occur on a regular basis. Writing in IEEE Spectrum, Charette [5] lists recent automotive recalls arising from software issues: the problems described include sudden shutdown and disabled passenger-side airbags.

There is also an emerging body of work demonstrating that software-controlled safety systems in automobiles can be compromised by malicious third parties. For example, Koscher et al [8] describe a suite of tests in which they used various attack vectors to successfully and substantially undermine safety systems in a commercially-available 2009 automobile. In closed course experiments they were able to successfully execute a number of disturbing attacks, including preventing the driver from braking, falsifying speedometer readings, and killing the engine.

ISO 26262 "Road vehicles – Functional safety"[3] is a new international standard, currently in the final draft (FDIS) stage. It adapts IEC 61508 to road vehicle E/E systems, including software components (safety-related and otherwise). Many auto manufacturers and their suppliers are already preparing their systems and processes for compliance.

While ISO 26262 does not specifically mandate any static analysis phases, incorporating static analysis can help simplify and improve the design, implementation, and testing stages of software development. This document describes how GrammaTech's static analysis tools CodeSonar® and CodeSurfer®

can be used together and separately to support an organization's ISO 26262 activities. They can be used throughout the product development phase, even before the software is ready to be tested. The cost of fixing a bug during development is much lower than that of finding one during testing, which in turn is much lower than the cost of fixing a bug that has persisted into a deployed product.

CodeSonar performs whole-program, interprocedural analysis on C and C++ source code, identifying programming bugs that can result in system crashes, memory corruption, and other serious problems. It includes numerous workflow automation features, including an API for custom integrations and support for extensions that add custom checks. CodeSurfer is a program-understanding tool that does precise source code analysis, calculating a variety of representations that can be explored through the graphical user interface or accessed through an optional programming API. CodeSonar finds bugs automatically, while CodeSurfer makes manual review of code easier and faster.

Typographical Conventions

The following typographical conventions are used in this document.

- CodeSonar warning class names are rendered in italic, sans-serif font: *Null Pointer Dereference*. If the warning class is disabled by default, the class name is marked with an asterisk: *Recursive Macro**.
- Direct quotes from the ISO/DIS 26262 document are formatted like the following:

  "quote"

Numbering

In general we will refer to numbered sections within the ISO/DIS 26262 document using the format

ISO 26262-P:C

Where P is the part number, and C is the (sub-)clause number within that part. For example, "ISO 26262-6:4.5" refers to sub-clause 4.5 of ISO 26262 Part 6 ("Product development: software level").

In contexts where the part number is clear, we may cite the (sub-)clause number only.

In order to minimize confusion between internal references and references to ISO 26262, the sections of this white paper are labeled with Roman numerals.

Terminology

ISO 26262 makes extensive use of a specialized vocabulary. In general, we assume that the reader is familiar with the terms and definitions used within the standard. For the sake of convenience, we reproduce below a small number of the key definitions from ISO 26262-1.

Automotive Safety Integrity Level (ASIL)

"one of four levels to specify the item's or element's necessary requirements of ISO 26262 and safety measures for avoiding an unreasonable residual risk with D representing the most stringent and A the least stringent level"

Cascading failure

"failure of an element of an item causing other elements of the same item to fail"

Dependent failures

"failures whose probability of simultaneous or successive occurrence cannot be expressed as the simple product of the unconditional properties of each of them"

Freedom from interference

"absence of cascading failures between two or more elements that could lead to the violation of a safety requirement"

Independence

"absence of dependent failure between two or more elements that could lead to the violation of a safety requirement; or organizational separation of the parties performing an action"

Inspection

"systematic examination of work products, following a formal procedure, in order to detect anomalies"

Organization

The main part of this white paper is organized according to major themes that cut across different parts of ISO 26262 and that can be addressed by GrammaTech static analysis tools. An appendix lists the individual parts, clauses, and subclauses of ISO 26262 that are discussed in this white paper together with page numbers for easy reference.

The use of CodeSonar and CodeSurfer is most applicable to ISO 26262 Parts 6 ("Product Development: software level") and 8 ("Supporting processes"), so these are the parts most heavily referenced in this paper. Parts 4 ("Product Development: system level") and 9 ("ASIL-oriented and safety-oriented analyses") also describe issues that can be addressed or partially addressed by these tools, and are referenced accordingly. Part 1 ("Vocabulary") is referenced when required for clarity.

Contents

I ASIL Management

A fundamental concept in ISO 26262 is the Automotive Safety Integrity Level (ASIL). Every safety requirement is associated with an ASIL, with A being lowest and D being highest; functionality that implements or supports a safety requirement is accordingly assigned an appropriate ASIL. In general, a software component with a higher associated ASIL will have more rigorous recommendations at each stage of the development process than a component with a lower ASIL.

The standard is highly concerned with the relationships between components and their ASILs, specifying how the relationships should arise and how they should be traced. CodeSurfer and CodeSonar provide mechanisms for documenting, maintaining, and verifying these relationships in various contexts. CodeSonar in particular can store ASIL notations at the warning report, analysis, and project levels. Furthermore, it can readily be extended to automatically compute and apply the appropriate ASIL notations to individual problem warnings.

By default:

- The ASIL of a software component is the highest ASIL of a safety requirement allocated to that component (ISO 26262-6:7.4.9).
- If the software components of the system have different ASILs, all software components will be "*treated in accordance with the highest ASIL*" (ISO 26262-6:7.4.10).

Exceptions to these defaults are described in I.2 *ASIL Decomposition* and I.3 *ASIL Coexistence*.

I.1 ASIL Tracking

Since each software component has an associated ASIL, it is highly desirable to label the analysis artifacts generated for a given component with the ASIL of that component. This enhances traceability and aids in prioritizing work items that arise as a result of the analysis. CodeSonar provides several powerful mechanisms for applying such labels.

When a software component is analyzed with CodeSonar:

- The analysis can be *annotated* with the component ASIL, either immediately or retroactively.
- A user can manually annotate all warnings issued by the analyses with the ASIL (all at the same time, if desired).
- A custom *warning processor* can automatically annotate each warning issued by the analysis with the ASIL.

The flexible search in CodeSonar includes functionality that allows users to find all warnings that have a particular ASIL annotation, or that were issued by an

analysis with a particular ASIL annotation. For example, a user might search for all *Buffer Overrun* warnings in components with ASIL D, or all warnings in components with ASIL B or higher.

Projects may contain components with different ASILs (as described in I.2 *ASIL Decomposition* and I.3 *ASIL Coexistence*). CodeSonar can provide ASIL tracking even in this case: the only differences are that the analysis annotation will need to include all relevant ASILs, and the warning processor (or person) responsible for annotating warnings will need to take each warning's location into account when assigning its ASIL.

I.2 ASIL Decomposition

ISO 26262-9:5 describes the mechanisms by which a safety requirement can be decomposed into multiple redundant safety requirements with "potentially lower" ASILs. This applies both to system design (ISO 26262-4:7) and software architectural design (ISO 26262-6:7). CodeSurfer and CodeSonar can be used to support the decomposition itself, and CodeSonar can accommodate ASIL annotation changes that may arise as a consequence.

Decomposition is closely tied to the notion of *independence*. The decomposition schemes depicted in ISO 26262-9:Figure 2 and described in -:5.4.7 are expected to result in independent elements; -:5.4.8 and 5.4.9 require that this independence be justified. The data and control flow analyses provided by CodeSurfer and CodeSonar and discussed in II *Control and Data Flow* are well-suited to furnishing evidence of independence.

ASIL decomposition can take place in advance of any CodeSonar analyses of the software, in which case the CodeSonar annotations and warning processors described in *I.1 ASIL* can be set up as using the decomposed ASILs from the outset. If some CodeSonar analysis has already taken place, the effects of requirements decomposition are communicated by simply updating the set of warning processors applied to the CodeSonar analysis. Similarly, if the hub includes previously-existing CodeSonar warnings that are now associated with newly decomposed requirements, custom warning processors can be used to update their properties appropriately.

I.3 ASIL Coexistence

ISO 26262-9:6 describes criteria by which a sub-element may be permitted to have a lower ASIL than its parent, or even no ASIL at all. II.2 *Coexistence of Elements* describes how GrammaTech static analysis tools can be used to inform these coexistence determinations, then subsequently to ensure that the properties required for coexistence are exhibited by the software.

As with ASIL decomposition (I.2), any changes to software component ASILs arising from a coexistence determination can be propagated into the existing hub database using custom warning processors and reflected in future analyses by

updating the warning processor that is responsible for computing and applying ASIL annotations.

II Control and Data Flow

Correct control and data flow are emphasized throughout ISO 26262. Software unit design and implementation is expected to have *"correctness of data flow and control flow between and within the software units"* (ISO 26262-6:8.4.4). Flow-based properties *partitioning* and *order of execution* are also specifically referenced at multiple points. Architectural requirements (ISO 26262-4:7.4.2) include compliance with ASIL decomposition and element *coexistence* criteria that are specified in ISO 26262-9:5 and -:6; these are also based in large part on control and data flow properties.

Control flow analysis and data flow analysis at source code level are strongly recommended methods for software unit design and verification at ASILs C and D, and recommended at ASILs A and B (ISO 26262-6:Table 10).

CodeSurfer provides powerful functionality for visualizing data and control flow in software, and for answering complex queries about both. Users can compare the flow computed by CodeSurfer to that defined in the software architecture to identify any discrepancies. In addition, users can find 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?" CodeSurfer's pointer analysis means that pointer aliasing and indirect function calls are accurately accounted for.

The CodeSurfer user interface and API allow 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.

II.1 Partitioning

ISO 26262 mentions software partitioning primarily as a means to establish "independence" and "freedom from interference". The requirements and recommendations for software development expand on the necessary properties of a partitioning scheme that implements freedom from interference (ISO 26262-6:7.4.11); -:Annex D (informative) provides a substantial introduction to partitioning from this perspective. The scope for analysis of dependent failures (ISO 26262-9:7) thus includes software partitions, and the system design specification is required to *"include the results of decisions concerning allocation and partitioning"* (ISO 26262-4:7.4.5.2).

At the software level, partitioning can be regarded as a flow-based property: to demonstrate that two software components belong to different partitions it is necessary to show that there is no mechanism allowing control or data to flow between them.

The CodeSurfer GUI and API provide powerful built-in queries, such as *slicing*, that can be used to check the integrity of partitioning schemes. The optional Path Inspector™ extension can be invoked to confirm that execution flow from one part of the program cannot reach another part (and, if necessary, vice versa). Similarly, custom CodeSonar checks can automatically detect many forms of partitioning breach.

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.

II.2 Coexistence of Elements

ISO 26262-9:6 provides “Criteria for coexistence of elements”, in particular for coexistence of software elements with different ASILs (including no ASIL at all). Coexistence determinations are based in large part on demonstrating *freedom from interference*, where an element A *does not interfere* with another element B if no failure of A can cause B to fail.

Given a software component C with safety-related subcomponents S1 and S2, and non-safety-related subcomponent N1:

- N1 can be “treated as a QM sub-element” with no associated ASIL only if it does not interfere with S1 or S2 *and* has no functional dependency with any safety requirement allocated to C. (-:6.4.3)
- S1 can have a lower ASIL than S2 only if it does not interfere with S2 for all safety requirements allocated to C. (-:6.4.4)

The abstract execution technology used in GrammaTech’s static analysis suite is extremely well suited to checking properties like these.

In some cases a non-interference determination will be trivial at the design level: it will be clear that S1 cannot interfere with S2 because it does not influence S2 in any way whatsoever. In other cases, deeper analysis is required. Once an element hierarchy and appropriate ASIL settings have been established, static analysis tools can be used to help check and enforce non-interference requirements. For example, suppose that a coexistence determination has been made on the grounds that software component X does not interfere with component Y. An

engineer can then create one or more custom CodeSonar checks to trigger warnings whenever interference is observed.

II.3 Order of Execution

Order of execution is a control-flow property, and *correct* order of execution is a necessary property in software architectural design (ISO-26262-6:Table 4) and software unit design/implementation (ISO 26262-6:8.4.4).

Many built-in CodeSonar checks are fundamentally related to correct order of execution. These include *Double Close*, *Use After Free*, *Return Pointer to Freed*, *Socket in Wrong State*, and many more. The CodeSonar API is also particularly well-suited to creating custom checks for violations of specific ordering constraints.

The Path Inspector extension to CodeSurfer can also be used to check execution order: given a specification of a desired order, a Path Inspector *Precedence* query will either present a counterexample path in which the order is not upheld or conclude – since no such path was found – that the ordering is always correctly maintained.

II.4 Supporting Safety Analyses

Safety analyses (ISO 26262-9:8) are expected to identify "*conditions and causes, including faults and failures, that could lead to a violation of a safety goal or safety requirement*".

Suppose we want to determine which parts of the program are affected by the execution of some function `f()`. CodeSurfer provides several ways to explore this question at different levels of focus. These include: forward slicing from the entry point of `f()` to identify the program points that are affected by the execution of `f()`, forward slicing from the value returned by a particular call to `f()` to visualize the effects of that call in isolation, and using a call graph viewer - with or without synchronized property sheets - to interactively explore the call tree from `f()`.

Conversely, CodeSurfer backward slicing, call graphs, and property sheets allow us to determine which parts of the program can influence the execution of some a function `g()`. More generally, slicing (and other queries), call graphs, and property sheets (and other UI features) can be applied to arbitrary sets of program points: not just functions.

III Architectural/Design/Coding Principles and Properties

Part 4: Product development: system level specifies "Measures for the avoidance of systematic failures" (ISO-26262:4:7.4.3), which include "Use of well-trusted design principles" (-:7.4.3.4) and a number of "modular design properties" (-:7.4.3.5). In particular, "hierarchical design" and "avoidance of unnecessary

complexity of HW components and SW components" are required for ASILs C and D, and recommended for ASILs A and B.

ISO 26262-6:Table 9 lists ten software unit design/implementation principles, all of which are supported by built-in CodeSonar checks or can be addressed by custom checks as shown below.

"*1a) one entry and one exit point in subprograms and functions*"

Can be addressed by a custom CodeSonar check.

"*1b) no dynamic objects or variables, or else online test during their creation*"

Partially addressed by the *Dynamic Allocation After Initialization** warning class. Can be more fully addressed by a custom CodeSonar check.

"*1c) initialization of variables*"

Directly addressed by the *Uninitialized Variable* warning class.

1d) No multiple use of variable names

Can be addressed by a custom CodeSonar check.

"*1e) Avoid global variables or else justify their usage*"

Can be addressed by a custom CodeSonar check. This principle is also supported by CodeSurfer, whose user interface provides direct access to a program's global variables.

"*1f) Limited use of pointers*"

Some limitations on pointer use are addressed by the *Function Pointer**, *Macro Uses -> Operator**, *Macro Uses [] Operator**, *Macro Uses Unary * Operator**, and *Pointer Type Inside Typedef** warning classes. Further limitations on pointer use can be imposed using custom CodeSonar checks.

"*1g) No implicit type conversions*"

Can be addressed by a custom CodeSonar check.

"*1h) No hidden data flow or control flow*"

Some restrictions on hidden data flow and control flow are addressed by the *Function Pointer** warning class, and by the various classes that limit the use of the preprocessor. Further limitations can be imposed using custom CodeSonar checks.

1i) No unconditional jumps

Directly addressed by the *Use of setjmp*, *Use of longjmp*, and *Goto Statement** warning classes. More subtle violations can be detected by *Empty if Statement*, *Empty switch Statement*, and *Redundant Condition*.

"1j) No recursions"

Directly addressed by the *Recursion* warning class.

III.1 Avoiding Unnecessary Complexity

Avoidance of unnecessary software complexity is stressed in both system design (ISO 26262-4:7.4.3) and software development (ISO 26262-6:8.4.3).

The "[p]rinciples for software architectural design" in ISO 26262-6:Table 4 include several for which satisfaction can be measured by standard code complexity metrics. CodeSurfer computes and reports a number of metrics that have become standard in software engineering practice, including the McCabe[10] and Halstead[6] metrics. CodeSonar will incorporate this functionality in the near future. The metrics are computed at the project, file, and function levels, allowing designers to easily identify problem areas.

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.

From a legibility/maintainability standpoint, it is often desirable to impose restrictions on the complexity of logical or numerical expressions. The AST access provided by the CodeSonar API is useful here. Given a specified limit on some aspect of expression complexity, a user can create a custom check that issues a warning whenever the analysis encounters an expression whose complexity exceeds this limit. Maintainability issues are covered in the next subsection.

III.2 Maintainability

Maintainability is recommended for software design and implementation at all ASIL levels (ISO 26262-4:7.4.3.5; ISO 26262-6:8.4.3).

In some cases source code may have straightforward underlying semantics but still be difficult for human readers to understand and therefore difficult to maintain. CodeSonar provides checks for code constructs that are particularly likely to impede clarity. These include occurrences of:

- Non-obvious flow of control *Function Pointer*, Use of longjmp*, *Goto Statement**,...
- Syntactic complexity: *Function Too Long**, *Too Many Dereferences**,...

- Hidden pointers: *Macro Uses ## Operator**, *Pointer Type Inside Typedef**,...
- Preprocessor directives (other than simple ones): *Conditional Compilation**, *Variadic Macro**, *Macro Uses ## Operator**,...
- Implied but nonexistent branching: *Empty if Statement*, *Empty switch Statement*, *Redundant Condition*. Note that warnings of these classes can also indicate that an error is preventing expected branching from occurring in practice.

Users can create custom CodeSonar checks for violations of other complexity restrictions that might be included in a project's design requirements.

III.3 Verifiability

Verifiability is an important goal for both system and software architecture design (ISO 26262-4:7.4.2.1; ISO 26262-6:7.4.2). Designing for verifiability includes avoiding unnecessary complexity where possible, as described in section III.1. Other CodeSonar support for verifiability is discussed here.

The presence of unbounded recursion or unbounded loops is generally an obstacle for verifiability. While 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.

Also desirable for the sake of verifiability is the ability to test and analyze code without altering it. Both CodeSonar and CodeSurfer observe the normal build process for a software project and use 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.

III.4 Coding Guidelines

The standard recommends topics to be addressed by design and coding (ISO 26262-6:5.4.6, Table 1). Once a set of C/C++ coding guidelines has been selected, CodeSonar can be used to enforce those guidelines. CodeSonar ships with built-in checks for some standard rule sets, such as Power of Ten[7]. For other coding guidelines, CodeSonar provides a rich API that allows users to implement custom checks.

At ASIL D, all of the topics listed in the standard are strongly recommended. At ASILs B and C they are all either recommended or strongly recommended.

CodeSonar support for coding guidelines that address these topics is described below.

"1a Enforcement of low complexity"

As described in III.1, CodeSurfer and CodeSonar support checking and enforcement of complexity restrictions in a number of different ways.

"1b Use of language subsets"

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

"1c Enforcement of strong typing"

The Table 1 notes include the remark that *"the objective of method 1c is to impose principles of strong typing where these are not inherent in the language"*. While there is no single universal definition of "strong typing", it is generally agreed that C and C++ are "less strongly typed" than some other languages (and thus certainly less so than a Platonic ideal) due to their support for implicit and explicit casting. Some coding guidelines, such as MISRA-C:2004[1], define a more type-safe version of the language by imposing restrictions on both forms of casting. CodeSonar supports such restrictions in two ways. Firstly, built-in checks for classes *Cast Alters Value*, *Dangerous Function Cast*, and *Varargs Function Cast* will issue warnings whenever certain type-unsafe casting phenomena are encountered. Secondly, users can write custom checks that issue warnings whenever specified type-unsafe operations are carried out. For example, if casting is completely forbidden, a check could inspect the ASTs for the project and issue a warning for *every* explicit cast operation. Checks based on AST inspection can also detect explicit casts between specific types, and implicit casts.

"1d Use of defensive implementation techniques."

CodeSonar provides numerous checks for code that violates defensive implementation principles. Furthermore, many of these principles are incorporated into the overall CodeSonar analysis at a fundamental level.

- Built-in warning classes *Scope Could Be File Static** and *Scope Could Be Local Static** enforce minimal scope for variables.
- Warning classes *Ignored Return Value* and *Unchecked Parameter Dereference** reflect a defensive perspective: functions should

check that their parameters are valid before using them, and their callers should handle all possible return values.

- Some aspects of library function failure behavior are accounted for in the CodeSonar analysis. For example, its default behavior is to issue a *Null Pointer Dereference* warning if a value returned by `malloc()` is dereferenced without being properly checked, because `malloc()` returns NULL on failure.

- In extreme cases, third party functions may be considered so problematic that their use is forbidden outright. CodeSonar provides a simple mechanism that allows users to specify functions whose uses should be flagged. Checks for uses of certain especially weak or error prone functions are shipped with CodeSonar and enabled by default. Examples of such functions are `crypt()` (which does not deliver the degree of encryption that its name might imply) and `gets()` (which is always vulnerable to buffer overflows). Checks for many other functions are shipped but must be specifically enabled; these include a substantial body of checks in support of DHS Build Security In (BSI) [2] coding rules.

- Configuration options allow users to tune the defensiveness of the CodeSonar analysis. In many cases the default settings of these options are the most defensive ones: for example, the default interpretation of a read through a volatile type is that it results in an unknown value.

- CodeSonar treats all external input as potentially adversarial. This includes the values returned by library functions like `scanf()`, and is orthogonal to any concerns about the correctness of the functions themselves. The assumption of adversarial input is also reflected in *Format String* warnings. Furthermore, the CodeSonar API provides a mechanism for modeling adversarial values. For example, a user could in effect instruct the analysis to treat the value returned by a particular function as adversarial, and do so without modifying production code.

"1e Use of established design principles"

This category is very broad, but many design principles are related to properties that are statically checkable. For example, design guidelines may impose restrictions on the use of global variables, or on exception handling: It is straightforward to write custom CodeSonar checks that inspect the internal representation and issue warnings whenever the relevant artifacts are observed (or observed outside permitted contexts).

"1f Use of unambiguous graphical representation"

This recommendation is not applicable to programming languages such as C and C++.

"1g Use of style guides"

Style guides typically cover topics such as code layout, capitalization, comments, and whitespace. CodeSonar includes several checks that support style enforcement, including *Function Too Long**, *Not Enough Assertions**, and *Too Many Dereferences**. The various thresholds associated with these checks (for example, the maximum permissible length for a function) are all user-configurable at multiple levels of granularity.

Custom checks for style guide violations are generally straightforward to write. For example, it is a simple matter to write a CodeSonar plug-in that issues warnings for all declarations of variables whose names include capital letters, or one that issues a warning for every source file that does not contain enough comments – whatever the definition of "enough" might be.

"1h Use of naming conventions"

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 can easily handle naming conventions that impose, for example, one set of rules governing the permissible names for variables and another for functions.

III.5 Robustness

ISO 26262 invokes the notion of "robustness" at several points. In the context of software unit design and implementation, robustness is deemed to include prevention of "*implausible values, execution errors, division by zero, and errors in data flow and control flow*" (ISO 26262-6:8.4.4). Robustness features noted for software unit testing (-:9.4.2) and software integration and testing (-:10.4.3) include "*absence of inaccessible software, effective error detection and handling*".

- Several built-in CodeSonar checks are concerned with implausible values. The majority of these, including *Function Call Has No Effect*, *Negative Character Value*, *and Unreasonable Size Argument*, deal with cases where certain argument values are deemed implausible for particular functions. Others, such as *Cast Alters Value* and *Integer*

Overflow of Allocation Size, identify explicit or implicit operations that may generate unexpected values.

- Division by zero is directly addressed by the CodeSonar warning class of the same name.
- CodeSurfer and CodeSonar both provide significant support for data flow and control flow analysis (discussed in II) and for identifying inaccessible software (discussed in VI.3).
- Effective error detection and handling applies at several levels of granularity. At the level of a single function, one standard practice is to return an error code when errors occur, and then to check the return value whenever the function is called. CodeSonar supports this paradigm with *Missing Return Value* and *Ignored Return Value* checks.

IV Reuse

The understanding, use, and modification of existing software is a fundamental part of any software development effort. ISO 26262 acknowledges this, setting out guidelines for reusing software both with and without modifications (ISO 26262:6:7.4.7, -:7.4.8).

Two significant issues for users of previously developed software are *understanding* the software and *trusting* that it does not contain bugs and vulnerabilities. These issues are covered by "Qualification of software components" (ISO 26262-8:12), where the components in question can be either COTS software or software developed in-house. CodeSurfer assists the qualification process by providing mechanisms for 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 can be applied to previously developed source code just as easily as it can be 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.

For software reused with modifications, the modifications themselves must comply with ISO 26262. All the development support provided by GrammaTech static analysis tools is as applicable to software modification as it is to new development.

Similarly, any proven in use argument as described in -:14 must consider changes that have occurred since the use in question (-:14.4.4). The change management support provided by GrammaTech static analysis products and described in X *Change Management* can be applied to previously-developed software just as it can to software currently under development.

V Configuration and Calibration

For the purpose of ISO/DIS 26262, *calibration data* is defined as "data that will be applied after the software build in the development process" (ISO 26262:1:1.11), while *configuration data* is "data that is assigned during software build and controls the software build process" (-:1.16). ISO 26262-6:Annex C (normative) comprehensively addresses configuration and calibration concerns.

"To enable the system integration subphase in accordance with Clause 8 the following shall be made available...c) if the system uses configurations or calibration data the verification at the system or vehicle level shall provide evidence of compliance with the safety requirements for each configuration at implementation level or for every configuration that is intended for serial production at a generic level" (ISO 26262-4:7.4.11).

V.1 Configuration Data

Both CodeSonar and CodeSurfer (including Path Inspector) account naturally for configuration data. The internal representations constructed by CodeSonar and CodeSurfer 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.

ISO 26262-6:Annex C:4.2 specifies that "*Verification of the configuration data shall be performed to ensure: a) the use of values within range;and b) the compatibility with values of the other configuration data*". Because the configuration data is built into the analyzed representation, CodeSurfer and CodeSonar analyses do not distinguish between configuration-dependent and configuration-independent data and control. Thus, if these analyses are used as a basis for verification, the established results apply to both.

If there are several valid configurations of the software, each can be analyzed separately to meet the requirements of ISO 26262-6:Annex C:4.4. This approach has several advantages. Firstly, the effects of concrete preprocessor directives, build settings, and other configuration data are directly reflected in the analyzed representation. Secondly, safety requirements that apply selectively to some configurations but not others can be addressed by checks that are carried out only by the corresponding analysis runs.

V.2 Calibration Data

As with configuration data, CodeSurfer and CodeSonar analysis do not distinguish calibration data from other data, so an analysis carried out on a calibrated piece of software will seamlessly incorporate the calibration-dependent components. Conversely, analyzing software that has been configured but not yet calibrated can help establish that desired relationships hold globally, and are not

dependent on specific calibration data sets. Ideally, then, the software will be analyzed in both uncalibrated and calibrated states.

An important area of concern is protection against unauthorized changes to calibration data (ISO 26262-6:Annex C:4.9, -:Table 1). GrammaTech static analysis tools are particularly well suited to determining whether the software allows calibration data to be modified, and to checking that protection mechanisms are used consistently. Given a particular use of a calibration data value, a user can perform a CodeSurfer *backward slice* to determine the program statements that can influence the value at that point. Path Inspector *precedence* queries help to verify that modifications to calibration data are always followed by consistency checks. If redundant storage is used to help protect the calibration data, a custom CodeSonar checks can be designed to trigger warnings at a designated point if the redundant stores are not synchronized at that point.

VI Test Support

Static analysis is a complement to testing, not a replacement for it. That said, static analysis tools can find many of the problems that would otherwise be found at a later stage by testing; more critically, they can detect problems that might be *missed* by testing with an insufficient test suite. Static analysis can also help resolve questions arising from test coverage analysis.

VI.1 Fault Injection

Fault injection testing is a recommended (highly recommended for higher ASILs) method for software unit testing (ISO 26262-6:Table 12) and software integration testing (-:Table 15), and for establishing that the system design specification and technical safety requirements have been correctly implemented for the sake of hardware-software (ISO 26262-4:Table 4) and system (-:Table 9; Table 12) integration and testing.

With the CodeSonar Extension API, users can carry out preliminary modeling of fault injection as soon as a code component compiles. Specifically, given a function `f()`, a user can write a model for `f()` that returns a deliberately-bad value, then instruct CodeSonar to divert all calls to `f()` to this model. Performing this modeling (and fixing the problems found) early in the development cycle can generally be expected to reduce the number of problems remaining by the time formal fault injection tests are carried out, which in turn will generally result in lower development costs.

VI.2 Resource Usage

In a similar vein, early static analysis can identify many resource usage problems, allowing them to be fixed in advance of the resource usage tests recommended for software unit testing (ISO 26262-6:Table 12) and software integration testing (-:Table 15), and for hardware-software (ISO 26262-4:Table 8) and system (-

:Table 13) integration and testing. CodeSonar includes built-in checks on resources such as pointers and file descriptors (*Leak*), memory (*Overlapping Memory Regions*), and the execution stack (*Excessive Stack Depth**).

VI.3 Reachability and Test Coverage

ISO 26262 makes structural coverage recommendations for tests at the software unit (ISO 26262-6:Table 14) and software architectural (-:Table 17) levels. 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 can be 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.

Code may be 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.

CodeSurfer 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 both CodeSonar and CodeSurfer work 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 examined.

VII Verification Support

> *"The first objective of verification is to ensure that the work products are correct, complete and consistent. The second objective of verification is to ensure that the work products meet the requirements of ISO 26262."* (ISO 26262-8:9.1)

Static code analysis and source code inspection are strongly recommended methods for verifying software unit design and implementation at ASILs B, C, D, and recommended at ASIL A (ISO 26262-6:Table 10, -:Table 11). Inspection is considered an "informal" method, while static code analysis is not. Static analysis

with both CodeSonar and CodeSurfer is highly applicable here, with CodeSurfer also providing substantial functionality in support of code inspection.

The verification of software unit design and implementation is expected to demonstrate that various properties hold, including compliance with coding guidelines, and compatibility with target hardware (ISO 26262-6:8.4.5).

VII.1 Compliance With Coding Guidelines

CodeSonar support for coding guideline enforcement is discussed in III.4.

VII.2 Compatibility With Target Hardware

To a substantial degree, handling for hardware compatibility concerns is built into the GrammaTech static analysis suite, because each analysis is carried out with respect to a specific software build, which in turn is targeted to a particular platform. Target-specific information such as address size and character signedness is built right into the GrammaTech internal representation of the project and taken into account when the representation is analyzed.

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

VII.3 Correctness and Consistency

The software is expected to correctly implement the system design, as made explicit in the objectives for item integration and testing (ISO 26262-4:8.1) 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, Double Initialization*
- *Unused Value*
- *Deadlock, File System Race Condition*
- *No Space for Null Terminator*
- *Return Pointer to Freed*, *Return Pointer to Local*

VIII Documentation

Documentation is addressed in ISO 26262-9:10. The CodeSonar hub database represents a substantial source of documentation material, and the CodeSonar user interface provides multiple output options for this material. CodeSurfer can also produce a number of documentary artifacts, including call graphs, system dependence graphs, and query result sets, but CodeSonar's documentation support is more comprehensive.

Documentation of analysis results is an integral part of CodeSonar. Each analysis of a project issues a number (possibly zero) of warnings. 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. 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 basic unit of documentation in CodeSonar is the Warning Report, which provides extensive information about a single warning issued by the CodeSonar analysis. As required by -:10.4.3, Warning Reports are precise, concise, clearly structured, easily understood, and maintainable. The formal elements required by -:10.4.3 are accounted for except "author and approver", as shown below.

- *"a) the title, referring to the scope of the document"* is provided by the report heading, which specifies the class and location of the warning. If further identification is required,
- *"b) the author and approver"* is not represented by default, unless one takes the author and approver to be the analysis itself. Users could, however, add this information by repurposing the Owner field of the report to record an author/approver pair, or by requiring that author and approver be recorded as part of the annotations on the report.
- *"c) unique identification of each different revision (version) of a document"* is provided by the Warning ID, which comprises a Report ID shared by all instances of "the same" warning and an Instance ID which is unique to the given instance.
- *"d) the change history"* is recorded as a series of event notations in the report Notes field. Links to reports for other instances of the warning are also provided and represent a change history of sorts – the history of the warning across multiple analyses of the project.
- *"e) the state"* can be recorded in the report State field.

For projects involving distributed development (ISO 26262-8:5), suppliers might themselves use GrammaTech static analysis tools. CodeSonar's hub-based architecture supports the sharing of analysis results between supplier and customer, maintaining a documentation history across organization boundaries.

IX Traceability

CodeSonar provides substantial support for traceability. Annotations can be applied to analyses and to individual warnings, becoming part of the historical record stored in the CodeSonar hub. Custom warning processors can automatically generate and apply such annotations. The annotations are

searchable, so users can find all warnings that have a particular annotation, or that were issued by an analysis with a particular annotation.

Two ISO 26262 phenomena that particularly need to be traceable are *ASILs* and *safety requirements*. Traceability for ASILs is discussed in I.1*ASIL Tracking*; we discuss traceability for safety requirements here. The same techniques can be applied to trace other information, if required.

Safety requirement specification and management is covered in ISO 26262-8:6. Safety requirements have unique, immutable identifiers (-:6.4.2.5) and are expected to be traceable (-:6.4.3.2). This traceability can be propagated into the CodeSonar analysis, and thence into the CodeSonar hub, by annotating warnings with corresponding requirement identifiers where appropriate. Custom warning processors can automate this process in a number of cases, including the following.

- A particular source code artifact or structural element (file, function, etc) is associated with a specific safety requirement.
- A particular warning class is associated with a specific safety requirement: this applies whether the warning class is built in or custom-written.
- Safety requirement unique identifiers have been incorporated into the source code supporting those requirements, perhaps by using delimiting comments or by incorporating the identifiers into function names.

X Change Management

Real-world software development is iterative in nature. ISO 26262 acknowledges this, and sets out requirements and recommendations for change management in ISO 26262-8:8. Both CodeSonar and CodeSurfer are designed to support this aspect of development.

X.1 Impact analysis

> "*An impact analysis on the existing system and its interfaces and connected systems shall be carried out for each change request*" (ISO 26262-8:8.4.3.1).
>
> "*Prior to release, if the change carried out has an impact on safety-related functions, the assessment of functional safety shall be updated*" (ISO 26262-8:8.4.5.2).

In general, modifications to a program can affect areas far removed from the site of the modification, and sometimes in unexpected ways. CodeSurfer provides powerful tools for identifying and understanding these effects. Given a program

region that has been modified, added, or removed, CodeSurfer users can obtain answers to questions such as:

- Which statements are affected by the changed region, and which statements affect it?
- How do the modifications affect the program call graph?
- How do control and data flow to and from the modified region?

The Path Inspector extension provides further support for impact analysis, allowing users to issue complex queries about execution flow in the modified program. For example, a user might invoke Path Inspector to confirm that execution flow between two parts of the program does not pass through the modified region, or that the modified region conforms to initialization requirements.

CodeSonar and CodeSurfer support re-analysis of areas affected by changes in several important ways. Firstly, both products can be 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 examining only those affected areas. Secondly, CodeSonar and CodeSurfer both support 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. Thirdly, 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).

X.2 Traceability

Each change request has an associated unique identifier (ISO 26262-8:8.4.3.1). If desired, the CodeSonar analyses and warnings associated with a change request can be manually or automatically annotated with the corresponding identifier, as described in IX *Traceability*.

XI Configuration Management

> *"The first objective is to ensure that the work products, and the principles and general conditions of their creation, can be uniquely identified and reproduced at any time.*
>
> *The second objective is to ensure that the relations and differences between earlier and current versions can be traced."*
> (ISO 26262-8:7.1)

ISO/DIS 26262 Parts and Clauses Discussed in this White Paper

Reference List

1. *MISRA-C 2004 Guidelines for the Use of the C Language in Critical Systems*, 2004, The Motor Industry Software Reliability Association.

2. Build Security In, https://buildsecurityin.us-cert.gov/bsi/home.html.

3. *ISO/DIS 26262, Road Vehicles - Functional Safety*. 2009, International Organization for Standardization.

4. Broy, M., *Challenges in automotive software engineering*. In *International Conference on Software Engineering*. 1928. Shanghai, China. pp. 33-42.

5. Charette,R.N., *This Car Runs on Code.* in *IEEE Spectrum*. February 1, 9 A.D.

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

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

8. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S., *Experimental Security Analysis of a Modern Automobile.* In *IEEE Symposium on Security and Privacy*. 2010. Oakland, CA. pp. 447-462.

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

10. 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.

GRAMMATECH

GrammaTech, Inc.
317 North Aurora Street
Ithaca, NY 14850
Tel: 607.273.7340
www.grammatech.com