GRAMMATECH

ACCELERATING AUTOMOTIVE SOFTWARE SAFETY WITH MISRA AND STATIC ANALYSIS

TRUSTED LEADERS OF SOFTWARE ASSURANCE AND ADVANCED CYBER-SECURITY SOLUTIONS

WWW.GRAMMATECH.COM

INTRODUCTION

The MISRA C/C++ coding guidelines came about due to concern for safely using C and C++ in critical automotive systems. Since its inception in 1998, MISRA has become one of the most used coding standards in the automotive industry and has spread to other safety-critical devices in medical and industrial control. Static analysis tools are needed to properly use and enforce the standard, but not all tools are created equal. Sophisticated static analysis tools that provide support for the complex development process and perform more than simple syntax checking are desired in order to reduce risk, costs, and time to market.

SUPPORTING THE DEVELOPMENT PROCESS

Static analysis tools support multiple aspects of the software development process, from early code development to post-launch debugging and forensics. Our whitepaper Making Safety-Critical Software Development Affordable with Static Analysis describes this in more detail. The recurring theme is that static analysis plays a critical role in improving software quality, enforcing safe coding standards such as MISRA, and detecting defects and security vulnerabilities that are difficult to find during testing.

MISRA plays an important role in C/C++ development when applied to safety-critical automotive software. Enforcement of the standard is difficult to do manually, so static analysis is used to enforce the standard. It is important to note, though, that there is a significant difference between advanced static analysis that supports MISRA rule checking and simpler tools that perform rule checking alone. GrammaTech's CodeSonar is capable of supporting MISRA coding enforcement but also providing valuable error detection and security vulnerability analysis that goes above and beyond simple code syntax checking.

CodeSonar is integrated with software development tools such as Jira, Bugzilla, Lattix Architect, Eclipse, etc. These integrations allow seamless adoption of static analysis into an existing development process. Source code quality and standards compliance can be checked right at the developer's desktop before checking into the build system. Defects and vulnerabilities can be automatically assigned for review and remediation. Audits can be done at any time and results distributed to the development team.

IMPROVING SAFETY WITH MISRA AND STATIC ANALYSIS

The MISRA guidelines themselves acknowledge the importance of tools in successfully using the guidelines:

> *…in its favour as a language is the fact that C is very mature, and consequently well-analysed and tried in practice. Therefore its deficiencies are known and understood. Also there is a large amount of tool support available commercially which can be used to statically check the C source code and warn the developer of the presence of many of the problematic aspects of the language.*
>
> *[MISRA-C:2004 Guidelines for the use of the C language in critical systems]*

MISRA guidelines define a safer subset of C or C++ that should prevent many classes of errors. Following these guidelines does improve code quality and, in combination with modern development tools, rigorous testing and good software development practices should improve system safety as well (assuming the level of rigour remains the same throughout hardware and system development.) However, no coding standard is perfect and enforcing the rules isn't enough, by itself, to ensure better software safety. Moreover, without tool support (as acknowledged by MISRA's authors) the standard doesn't stand on its own.

To really understand why static analysis is crucial in producing safety-critical software for automotive applications, we need to first look at how defects get into programs, and second, how these defects are found and fixed. Defects are an unfortunate but unavoidable side effect of writing code. Research has shown that software developers introduce defects at an average rate of one defect per ten lines of code. Most development time is spent finding and fixing those defects.

But by the time the code is released, most commercial software will still have about one defect per 1,000 lines of code (KLOC). Open-source code is a little better, at 0.68/KLOC. Cleanroom software, which combines formal methods of requirements and design with statistical usage testing, was found to have 0.1/KLOC. Considering that the software content of today's luxury car is well above 100 million lines of code, even in the absolute best-case scenario, there would still be at least 10,000 latent software defects and, in practice, there are likely more than 100,000. The defects that remain in the software are typically difficult to detect in testing, and can result in poor interoperability with other subsystems or unchecked interfaces, unexpected behavior, or outright failure.

Finding and fixing defects during testing is extremely time consuming. When a tester finds an error or a failure, the root cause is in many cases unknown, forcing the developer to trace the problem back to source. This means that the circumstances that created the failure must be reproduced, and the developer must study and understand the related code. The reality is that many bugs cannot be reproduced on demand and are never fixed.

The power of static analysis is that it doesn't rely on test cases to find problems, nor does it require that an error or failure be reproduced. An advanced static analysis tool can infer the runtime behavior of a program without actually running the program. Furthermore, when it identifies a problem, it also pinpoints the locations within the code that created the failure. This makes the job of debugging far simpler. Static analysis does not eliminate the need for testing altogether, but it

complements testing activities, providing an additional and critical validation technique. The reality is that in large and complex software systems, there are so many possible states, and such an astronomical number of possible paths of execution, that it is infeasible to exhaustively test them all. Static analysis, on the other hand, can explore these paths and state conditions in the aggregate, and is able to find problems that are missed by testing.

DETECTING ERRORS THAT CODING STANDARDS AND TESTING MISSES

A key contribution that advanced static analysis tools like CodeSonar provide to safety-critical software development is the ability to find defects that have slipped through the traditional development techniques. As evidenced by high profile cases in the automotive industry, safety issues that make it to market and on the road are expensive to fix – orders of magnitude more expensive than during development. Additionally, critical defects can be detected early, even early enough to be caught during coding. Some examples of classes of defects that can be missed during development, even when good engineering practices, MISRA coding, and rigorous testing is done, are given below.

- **Concurrency defects** often occur randomly and only after a system has been integrated completely on the final hardware platform. Unlike other testing, GrammaTech's CodeSonar can reason about multithreaded/multitasking code behaviour and detect dead locks, race conditions, and other types of concurrency errors.

- **Security vulnerabilities** are software defects that can be exploited to interfere with a system's behavior or expose critical data. Security is often overlooked in systems where safety is paramount. Advanced static analysis tools can detect security vulnerabilities that arise from malformed or tainted data outside expected values.

- **Tainted data analysis** helps to trace input data in the system to its use in the application and warn of any potential security vulnerabilities that arise. In today's hostile operating environment, it's foolhardy to assume that system input data is well-formed. Detecting these types of security issues is very difficult when data is passed across many functions. Automated analysis can provide the full control and data path for tainted data, which allows for rapid remediation.

- **Complex inter-procedural defects** are difficult to detect, especially with unit and subsystem testing. CodeSonar does advanced inter-function (procedure) analysis of control and data flow of the entire scope of the program. Deep analysis decreases the rate of false positives (errors that are false) but also increases the rate of true positives (errors that are verified true). CodeSonar's analysis extends into executables, object files, and libraries.

- **Binary analysis** provides insight and error detection of compiled code as object files, libraries, and even executables. A unique capability of CodeSonar is that it performs the same detailed analysis on binary code as it does on source. This can include system and third-party libraries that are provided without source code. Developers can ensure that the all code (both binary and source) is up to the quality standard required for the project.

MISRA RULE COVERAGE

When developers are required to adhere to coding standards, they look for a tool that can help them find violations. One of the metrics they use to compare tools is the coverage (the proportion of rules that the tools claim to check), with a naïve strategy being to choose the tool that claims the highest coverage.

Unfortunately, the notion of coverage is not well-defined, and because there is no reliable source of information that can be used to compare coverage between tools, customers must trust vendors to reasonably interpret the term, and to report their coverage fairly.

SOME COVERAGE IS EASY

Some MISRA rules are simple enough that a straightforward code-syntax-checker can find all violations with no false positives. For such rules, coverage is easy—the tool can either find violations or not, with no ground in-between. In MISRA C 2012, such rules are labeled Decidable. If the violation can be detected by looking only at one compilation unit, the rule is also labeled Single Translation Unit. For example, rule 4.2 forbids the use of Trigraphs, and is labeled this way. Claiming coverage for this rule is believable and provable.

Less clear, however, is if a violation can only be reliably found if the tool needs to look at multiple compilation units at the same time (these are labeled System in the standard). For example, MISRA C 2012 rule 5.1's "External identifiers should be distinct" is certainly decidable, but the only way a tool can reliably find a violation is if it examines all compilation units and compares all such identifiers found in each.

If a tool claims to have full coverage of a rule with System scope, then it is only reasonable to believe that the tool is also capable of finding all the compilation units that contribute to the program. Over-approximation and under-approximation of the set can lead to both false positives and false negatives. Humans routinely get this wrong, so a user of a tool that does not offer an automatic way to determine the set is running the risk of getting incorrect results.

The most effective tool is one that integrates tightly with the build system, as that is most often the most trustworthy source.

UNDECIDABILITY

In MISRA C 2012, some rules are labeled "Undecidable," meaning that it is fundamentally impossible to have a method that can, in general, say for sure if a violation is present or not. Because of this property, the author of a checker must find a sweet spot that balances the risk of false positives with the risk of false negatives. Most of these rules require an analysis that is capable of reasoning about the execution of the program, so only the most sophisticated static analysis tools can be expected to do a good job. A good example is MISRA C 2012 rule 17.2, which forbids recursion, both direct and indirect (i.e., calls through function pointers).

The trouble is, claims of coverage are often made that ignore whether a tool is good or bad at finding violations. If a tool can only find the most obvious and superficial instances of violations, is it reasonable for it to claim that it has coverage of that rule? The other side of that coin is interesting to consider too — if a tool finds all violations, but also reports so many false positives that is impractical to inspect them all, is it fair to say that it has coverage?

COVERAGE BREADTH

The final aspect of rule coverage that makes it complicated is that coding standards are usually quite loosely defined, whereas static analysis tools must have a precise definition of the properties that they are looking for. Consequently, it is common for a checker to detect a property that is either a superset or a subset of what the rule requires.

For example, let's consider CodeSonar's coverage of MISRA C 2012 Rule 2.2: "There shall be no dead code." For the purposes of this rule, dead code is code that is executed, but whose removal cannot affect program behavior. CodeSonar has an Unused Value checker, which finds places where a variable is assigned a value that is never subsequently used. All such places violate the MISRA rule, but there are other ways in which the rule can be violated that are not detected by this checker. Thus, the Unused Value checker covers only a subset of what the rule specifies, and other CodeSonar checkers fill in the gaps.

In some cases, the rule and the checker are not in a strict subset/superset relationship. They may overlap a lot or a little, or the checker may detect a property that is not a direct violation but is very likely to lead to a violation of the rule.

In CodeSonar, the policy is to claim coverage only if there is a large overlap between what the rule specifies and what our checker will find, and where the checker does not yield warnings that would be reasonably judged to be false positives for that rule (notwithstanding that they may be true positives otherwise).

MISRA C 2012, RULE 1.3

There is one MISRA C 2012 rule in particular for which the issue of claimed versus real coverage is acute. One should be wary of a static analysis tool claiming coverage of MISRA C 2012 Rule 1.3: "There shall be no occurrence of undefined or critical unspecified behavior." This rule is so broad that an additional 10-page Appendix in the standard outlines possible scenarios to avoid. This in turn references the C standards: those for C90/99 enumerate 230 instances of undefined behavior (65 of these are not covered by any other MISRA rule), and 51 instances of critical unspecified behavior (of which 17 are not covered by any other MISRA rule). Furthermore, the rationale for Directive 4.1 adds: "the presence of a run-time error indicates a violation of Rule 1.3."

Consequently, Rule 1.3 specifies an enormous amount of forbidden behavior, including null pointer dereferences, buffer overruns, use of uninitialized memory, data races, use after free errors, and many of the other hazards of programming in C.

The problems with rule coverage claims should be clear — although this one rule (of the 143 in the standard) constitutes only 0.7% of the standard, it covers maybe 50% of the truly unpleasant kinds of failures that can befall a C program.

Furthermore, if a tool is to claim coverage of Rule 1.3, it should have checkers that have a good-sized intersection with all of those undesirable behaviors. If a tool can only find a tiny percentage of them, then it is unreasonable for it to claim coverage. In contrast, advanced static analysis tools such as CodeSonar are explicitly designed to find the kinds of run-time errors that constitute violations of Rule 1.3. Analyses that make it possible for them to find such defects with reasonable precision must be whole-program, path-sensitive, aware of hazardous information-flows, and capable of reasoning about concurrently-executing threads.

ACCELERATING TIME TO CERTIFICATION

Obvious advantages of static analysis tools include risk-reduction and time-reduction from finding and fixing defects and vulnerabilities in early stages of development. The cost savings of finding critical defects is significant compared to finding and fixing these bugs in system integration or worse, when products are in service. Additionally, CodeSonar provides automated documentation to support testing and quality/robustness evidence. Much of the manpower used in safety certifications is documentation and evidence production. Automation, and specifically static analysis, reduces this burden significantly. As a TUV SUD certified ISO 26262 tool, CodeSonar provides assurances to developers that it can be integrated into a safety-critical development project without further certification requirements – something that adds costs and risk otherwise.

CONCLUSION

The tools needed to support successful safety-critical projects require more than simple source analysis and MISRA rule checking. Enterprise-level development projects require sophisticated tools that support and enhance the development lifecycle, including integrating with other development automation tools. The ability to go beyond MISRA rule enforcement and prevent critical defects and vulnerabilities from leaking through the development process pays big dividends in cost and risk reduction.

REFERENCES

Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013.

“Making Safety-Critical Software Development Affordable with Static Analysis”, GrammaTech whitepaper, 2016.

“Advanced Driver Assistance Systems (ADAS), Safety, and Static Analysis”, GrammaTech whitepaper, 2016

“Reduce Automotive Software Failures with Static Analysis”, GrammaTech whitepaper, 2015

“How to Avoid Common Pitfalls in MISRA Compliance”, GrammaTech whitepaper, 2014

GrammaTech, Inc. is a leading developer of software-assurance tools and advanced cybersecurity 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.