GRAMMATECH

REDUCE AUTOMOTIVE SOFTWARE FAILURES **WITH STATIC ANALYSIS**

TRUSTED LEADERS OF SOFTWARE ASSURANCE AND ADVANCED CYBER-SECURITY SOLUTIONS

WWW.GRAMMATECH.COM

INTRODUCTION

Today's automobiles run on an astonishing and increasing amount of networked software, much of which powers safety-critical systems. Virtually every aspect of a car is controlled by software, including the throttle, transmission, brakes, speedometer, climate, lights, navigation, and entertainment. With well over 100 million lines of code running on 60 or more interconnected embedded computers, software defects and security breaches can impact safe operation, presenting a huge risk to human safety.

Unfortunately, the industry's growing innovation, complexity, and reliance on safety-critical software have caused recalls to skyrocket, with software defects accounting for 60-70% of today's recalls. Toyota's unintended acceleration flaw, for example, in its drive-by-wire throttle system, cost the company an estimated $5 billion in damages and lost revenues. A significant loss, the Toyota example is an important lesson in software safety – a reminder that software development must evolve, helping teams develop high quality software and break away from the "unaffordability wall."

THE INTRODUCTION OF SOFTWARE STANDARDS

Aware of the growing issues and importance of software safety, the automotive industry embraced a set of software development guidelines for the C/C++ programming languages called MISRA. These guidelines are intended to facilitate coding best practices, improving safety, portability, and reliability. Additionally, in 2011 the automotive industry defined the ISO 26262 international standard. Like its parent standard, IEC 61508, ISO 26262 is a risk-based safety standard, in which the risk of hazardous operational situations is qualitatively assessed and safety measures are defined, to avoid or control systematic failures and to detect or control random hardware failures, or mitigate their effects. The standard does the following:

- Defines an automotive safety lifecycle (management, development, production, operation, service, decommissioning) and supports tailoring the necessary activities during these lifecycle phases.
- Covers functional safety aspects of the entire development process (including such activities as requirements specification, design, implementation, integration, verification, validation, and configuration).
- Defines an automotive-specific risk-based approach for determining risk classes called Automotive Safety Integrity Levels (ASILS) and uses them for specifying what is necessary for achieving an acceptable residual risk.
- Provides requirements for validation, and confirmation measures necessary to ensure that a sufficient and acceptable level of safety is being achieved.

These standards are thoroughly defined. Given these efforts of the automotive industry to adopt standards and best practices, why is it still so difficult to produce high quality software? What can manufacturers do to make sure their software is safe?

SPECIAL CHALLENGES FACING AUTOMOTIVE SOFTWARE DEVELOPERS

The large-scale use of multi-function software in automobiles is a relatively new phenomenon, including subsystems that provide functionality typically aligned with entertainment and communications applications. Since the automotive beginnings, vehicle systems have been solely focused on well-defined single functions, relying on mechanical and analog systems that are easily tested. Software, especially for inter-connected systems, is far more complex than mechanical systems, and newly challenging for developers because of increased system scope, new skills and awareness of threats, and openness.

Software makes it possible to do things that otherwise would not be practical – for instance, fuel-consumption optimization, autonomous driving aids to improve safety, live updates, and more. Due to its complexity, not only is it difficult to write defect-free applications, but numerous other issues can plague software systems – incompatibilities between subsystems, increased cyber-threats due to the internet connectivity, race conditions that lead to system instability, etc. Ultimately, automotive software is complicated, with a huge burden being placed on teams that are new to the complexity of highly connected safety-critical software.

So the question remains, how can we be sure that a program will run correctly? And in the case of safety-critical code where human lives are at risk, how do you know whether a program is safe to use?

GETTING IT COMPLETELY "RIGHT"

A helpful approach to ensure safety in automotive software is to learn from the successes of NASA, Boeing, Lockheed Martin, and others who have dealt with extremely complex software systems that are mission and/or safety critical. NASA's Mars Curiosity mission, in which millions of lines of software were developed to control both spacecraft and rover missions, employed development best practices and higher verification techniques to eliminate risk of failure. This process included defining a good software architecture based on a clean separation of concerns, data hiding, modularity, well-defined interfaces, and strong fault-protection mechanisms.

NASA uses sound development processes, with clearly stated requirements, daily integration builds, rigorous software analysis and testing, and extensive simulation. According to Gerard Holzmann, senior research scientist at the NASA Jet Propulsion Laboratory (JPL), "Every precaution was taken to optimize the chances of success." In fact, for over 18 years, NASA's Independent Verification and Validation Program has utilized static code analysis tools to find defects that pose a risk to the success of NASA's most critical missions, assuring that software operates safely, reliably, and securely. The essence of NASA's software development approach, which can be applied to any safety-critical software application, is to (1) implement development best-practices, (2) adopt a coding standard, and (3) use the most sophisticated static analysis tools available to gain greater insight into the quality and security of the software.

THE BENEFITS OF STATIC ANALYSIS

Static analysis benefits have been known and utilized by most industries with safety-critical requirements, from commercial aviation supporting DO-178B/C certifications to industrial automation supporting IEC 61508 certification. MISRA C guidelines explicitly recommend the use of automated static analysis tools to find violations of the standard. Static analysis can be thought of as an automated code review in which another piece of software (rather than another person) inspects source code to find potential problems.

Static analysis tools can be used to find several types of issues, including violations of coding standards and rules, misuse of language APIs, concurrency issues, dataflow integrity issues, and security defects. Advanced static analysis tools, like GrammaTech's CodeSonar, have deep semantic knowledge of an entire program, enabling it to identify critical defects across the entire application without the need to create test cases. And by turning on MISRA or ISO 26262 checkers, CodeSonar easily identifies compliance defects.

STATIC ANALYSIS FOR AUTOMOTIVE

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 most of those defects.

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 critical valida-

tion 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.

HOW DOES STATIC ANALYSIS WORK?

Static analysis tools have become very sophisticated over the years, and while early tools considered only the structure or behavior of individual statements and declarations, today's advanced tools can include the complete source code of a program to perform whole program analysis. These tools work much like compilers, taking source and binary code as input, then parsing it and converting it to Intermediate Representations (IR). Whereas a compiler would use the IR to generate object code, static analysis tools retain the IR, which functions as a model of the program.

The effectiveness of the static analysis tool is determined by the level of sophistication of the algorithms that explore the model. The most sophisticated algorithms model the state of the program as a set of abstract equations about paths of execution through the entire program, across function and compilation unit boundaries. Static analysis tools are organized as sets of checkers that perform various types of analyses on the code. Checkers find specific kinds of defects and policy violations by traversing or querying the model, looking for particular properties or patterns that indicate problems. Using a symbolic analysis engine, static analysis tools explore program paths, reasoning about program variables and how they relate. Advanced dataflow analysis prunes infeasible program paths from the exploration. When the path exploration notices an anomaly, a warning is generated.

An enormous number of combinations of circumstances must be modeled and explored, so static analysis tools must employ various strategies to ensure scalability. For example, procedure summaries may be refined and compacted during analysis, and paths are explored in an order that minimizes paging. Through this process, static analysis is able to catch real bugs that are difficult to find by testing alone, including buffer overflows, memory leaks, uses of uninitialized data, null pointer dereferences, and concurrency errors.

CODESONAR'S ADVANCED ANALYSIS ENGINE

CodeSonar is one of the industry's most advanced static analysis tools, detecting over a hundred types of problems right out of the box. Because CodeSonar performs a unified dataflow and symbolic execution analysis that examines the entire program and doesn't rely on superficial pattern matching or similar approximations, it is able to find many types of defects that are missed by other static analysis tools.

Most automotive systems include third-party binary code, for which source code is not available. Research from VDC indicates that as much as 30% of embedded applications leverage third-party software, such as operating system graphics and connectivity toolkits, cryptography libraries, optimization libraries, and others. Because the integration of third-party software

can introduce defects, it is valuable to examine it. CodeSonar is the only static analysis tool that can work with embedded source and machine code, supporting complete system mixed-mode analysis. Mixed mode uses binary analysis to extract the semantics of the binary code, and uses it to present warnings in the parts of the source code that interact with the binary. Mixed analysis mode makes it possible to find defects coming from third-party code, which are invisible to other static analysis tools.

Some of the most difficult bugs to find are concurrency problems, and the prevalence of these is increasing as the use of multicore architectures accelerates. Special programming techniques are required to avoid concurrency problems such as race conditions, deadlocks, and resource starvation, and even very skilled programmers can make mistakes. CodeSonar's concurrency checkers for C and C++ are especially advanced, having been developed, tested, and commercialized over years of innovation. Today's concurrency checkers continue to excel, continually winning benchmark assessments, such as Toyota's Quantitative Evaluation of Static Analysis Tools published by Shin'ichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu.

Now that automobiles are connected to networks, including 6LoWPAN, Bluetooth, Wi-Fi and 4G, they are subject to the same threats as other connected devices. While not all defects lead to system failures, certain defects can expose a system's vulnerability to security attacks. CodeSonar uses a number of checkers and advanced analyses for identifying potential security risks in code, identifying Common Weakness Enumeration (CWE) instances, violations of US CERT guidelines, and tainted information flow. Taint analysis determines whether an input value comes from a risky source, such as a sensor or user input, and determines how that value can flow through the program to sensitive parts of the code. It checks if the value falls within the expected range, since an out-of-bounds value could lead to a system failure or could be exploited by an intruder to gain control of the system.

CodeSonar's automation will help automotive developers accelerate compliance with standards while reducing costs in many ways, including the following:

- Complying with coding standards requires auditing, documentation, and workflow tracking within the development process. While static analysis tools can't on their own ensure compliance with ISO 26262, they can, for example, aid developers in demonstrating compliance by addressing many of the requirements found in Part 6 of the standard (the section that covers "Product Development at the Software Level" for the functional safety of road vehicles, and examines correctness of software design and implementation).

- Verifying that the 143 coding rules found in MISRA C:2012, the latest standard, are checked and in compliance, is made possible with CodeSonar. See the GrammaTech white paper "How to Avoid Common Pitfalls in MISRA Compliance" – for more information about complying with the MISRA standard using CodeSonar.

- CodeSonar has also been certified for use in the development of safety-critical software up to the highest safety integrity levels for safety-related standards ISO 26262, EN 50128, and IEC 61508. It has been classified by the DO-178B guidance as a software verification tool, as defined in section 12.2 of the guidance.

INTEGRATING STATIC ANALYSIS WITH YOUR SOFTWARE DEVELOPMENT PROCESS

Integrating static analysis into your existing development process or enhanced SDLC process is simple and can be done at any stage. Furthermore, CodeSonar does not require any changes to the build process, build tools, or source code. Once installed and configured, it can run seamlessly any time code is compiled. This makes it possible to find bugs as they are introduced, when the cost of fixing them is least.

CodeSonar is suitable for both small and large scale embedded software projects, and can perform whole-program analysis on more than 10+ million lines of code. Additionally, CodeSonar has been designed to support concurrent analysis, so it can take advantage of the processing power of a cluster of loosely-coupled computers, dramatically increasing analysis time over most other static analysis tools.

CodeSonar helps automate team workflow and includes powerful tools for program analysis, program inspection, program understanding, and architecture visualization. It enables developers to discover the underlying design intentions of existing code, and recognize when new code deviates from this design. It provides warnings when new defects are first introduced and uses cutting edge technologies to help developers identify and understand them. CodeSonar was developed for mission-critical embedded software projects, both large and small. It offers an affordable, customer-friendly permanent licensing model that enables organizations to use the tools across projects.

CONCLUSION

Developing embedded software for safety-critical applications, such as automotive systems, requires a rigorous approach. Best practices, including process and coding standards, are proven methods for improving code quality. Advanced static analysis tools help automate enforcement of coding standards, and support standards compliance activities. More importantly, they play an essential role in finding defects in code that are missed during other V&V activities, and they aid developers in understanding and correcting code problems. Because it is infeasible to exhaustively test large and complex programs due to the overwhelming number of possible code execution paths, static analysis offers the only feasible means to fully explore and verify the software. When used in conjunction with other quality assurance practices, advanced static analysis tools can significantly reduce the risk of in-service failures. Don't place your customers at risk or inadvertently harm your company's brand and trustworthiness by deploying defective software systems – static analysis is an inexpensive, easy-to-deploy automation technology that will quickly deliver significant ROI.

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.