Software Static Code Analysis Lessons Learned©

Andy German
QinetiQ Ltd.

The United Kingdom Ministry of Defense has been in the forefront of the use of software static code analysis methodologies, including some of the tools and their application. This article[1] discusses what is meant by static analysis, reviews some of the tools, and considers some of the lessons learned from the practical application of software static code analysis when used to evaluate military avionics software.

Most software errors are relatively harmless, albeit annoying, such as when a word processor crashes. However, errors in some types of software can have serious consequences such as the failure of an aircraft's flight control software, which could be catastrophic. Software that controls a system whose failure could endanger human life or the aircraft is termed *safety-critical software*. Its integrity is of great concern to developers, users, the public, and the certification/regulatory authority.

Recent large-scale assessments of avionics software have produced some interesting results that show how important language selection is when producing safe and reliable avionics. This article presents the following information:

- Covers some of the methods used to identify safety-critical software and functionality.
- Discusses some myths of static code analysis.
- Describes some static analysis techniques.
- Identifies some of the tools available.
- Provides some general results of the practical application of static code analysis.

Static Code Analysis

Safety-critical software must be shown fully predictable in operation and have the properties required of it [1]. In addition to dynamic testing, such code is also subject to static testing: This is the rigorous examination of software (without running it dynamically) to establish the properties that will always hold true under any operating condition. It is an examination of the code, the architectural design, and the accompanying documentation, which together provides a picture of the completeness, or otherwise, of the software system [2].

There are various techniques that come under the umbrella term *static code analysis*, and these can be characterized by their nature and depth [3]. Nature refers to the broad objectives of the analysis and could be concerned with specific properties such as portability. Depth means the analytical depth of the technique.

Identification of Safety-Critical Software

The United Kingdom (UK) Ministry of Defense (MoD) adopted the safety argument approach in 1992, as retrospective evaluation of avionics systems had become complicated. The MoD still operates the *lessons learned/best practice* approach that is used as part of the safety argument evidence. The system design standards are used to trap system safety design requirements; these are Defense Standard 00-970 [4] and Defense Standard 00-971 [5] for aircraft. The safety argument approach is now used for the complete aircraft and has major advantages; it does not limit the possible design solution by being over-prescriptive, and it can cope with rapidly changing technology.

The current preferred method for safety-critical code functionality identification (including system robustness) is to use a top-down analysis starting with the defined safety targets for tolerable catastrophic mishap rates, including aircraft loss. Recent aircraft projects have shown that *bottom-up* hazard identification produces somewhere between 700 and 1,500 hazards. The bottom-up approach does not prioritize the hazards or show their relationship to the system as a whole; their true categorization is unknown. These large numbers of hazards are difficult to manage, and so a top-down evaluation is used to refine the argument.

The top-down approach normally results in approximately 100 of the most significant hazards being identified from approximately 10 top-level accidents/events. The Hazard and Operability (HAZOP) [6] approach to system and software functionality assessment has demonstrated itself to be an invaluable tool, particularly for defining system robustness requirements. This approach allows the system designer and regulatory authorities to show, through reasoned argument, that the following occur:

- Hazards are identified.
- Safety functionality is understood.
- Robustness requirements are identified.
- Hazards are mitigated to a tolerable level.

The first and best step in hazard mitigation is to avoid using safety-critical software wherever possible.

Why Use Static Code Analysis?

For UK defense projects, Defense Standard 00-55 [7] is normally recommended. This standard details two basic approaches to safety critical software:

- The use of formal methods (correct by design).
- The static analysis of the code (conformance with the design).

These are coupled with the following:

- Selection of a suitable high-level language (including its subset definition where appropriate).
- Defensive programming.
- Independence in verification and validation activities.
- Comprehensive documentation and configuration management.
- Testing and test coverage.
- Compiler validation.

The formal methods approach has not been widely adopted for the following reasons:

- Some of the most recent aircraft entering service started development back in the late 1970s when formal methods tool and support was severely limited. This is also prior to Defense Standard 00-55 initial issue requiring the use of formal methods.
- More recently, it is because of the short lead times and hence the extensive use of commercial off-the-shelf components for which the Civil Airworthiness Authorities do not mandate the use of formal methods. The

Civil Airworthiness Authorities suggest that the use of formal methods be considered [8].

The application of static code analysis techniques in retrospect is not ideal; the process is best suited and cheapest when applied during software development. Following the UK *as low as reasonably practical* approach [9] to risk, the retrospective evaluation of safety-critical code is the only reasonable method available at present to reduce safety-critical anomalies to a minimum – after all other mitigations have been considered.

Static Analysis Myths

But We Test It

All software contains errors, and computer programs rarely work the first time [10]. Usually, several rounds of rewriting, testing, and modifying are required before a *working* solution is produced [11]. Testing usually involves running and evaluating the software across its expected range of operation. This process is limited by the tester's ability to predict this range of operation, or rather, the range of inputs that the program will receive. This is how it is possible for large *well tested* software packages to still fail periodically: The user has done something not anticipated by the tester. It would be nice to test every state of a program, but such exhaustive testing is impractical as it would take far too much time and expense.

An argument often used against static analysis is that "our software has been extensively tested." This argument does not stand up. Radio Technical Commission for Aeronautics, Inc. (RTCA) Defense Order (DO)-178B [8] Level A requires extensive modified condition/decision coverage testing while RTCA DO-178B Level B does not require this level of testing. When Level A was compared to Level B, no significant difference in anomaly rates identified by static analysis was found. Unhappily, the *hack-it-and-bash-it* methodology is still prevalent among many software developers.

Static Analysis Means It Is Safe

The phrase "static analysis means it's safe" is heard quite often. Static analysis only allows us to argue that the code is as follows:

- As compliant with the software requirements as present evaluation methods and technology allows.
- That *coding errors* have been minimized.

Static analysis does not prove that the requirements the code was developed from were correct or show that the compiled code is correct.

It Costs Too Much

Based on project experience, an average 10 percent of a military aircraft's software – or approximately 500,000 software lines of code – are found safety critical. The average cost of retrospective independent analysis for an aircraft is less than $13 million, and on average less than 0.4 percent of the total development cost for an aircraft. These costs can be further reduced if the semantic analysis element is directed. It has been shown that the most costly element of static analysis is the semantic element when comparing costs of the activity to total percentage of anomalies found. One important area for future research is the justifiable targeting of techniques.

If, however, the software is designed and analyzed as part of the development process, then the cost savings are likely when compared to normal industry costs. There are also considerable through-life cost savings and system reliability benefits.

Dissimilar Systems

Although Defense Standard 00-55 [7] allows the use of dissimilar systems to be combined to create a safety-critical system, this becomes a very difficult approach to argue as being safe. The following issues need to be addressed:

- The comparison software or *liveware* (pilots) becomes safety critical (70 percent of aircraft accidents are due to aircrew error).
- How do you prove dissimilarity?
- Reliability goes down, as the lower integrity systems are likely to disagree and fail more often.
- The warning system becomes more critical.
- The cost of ownership goes up (supporting multiple equipment, increase aircraft weight, etc.).
- Designers tend to make the same mistakes.

Main Static Analysis Techniques and Methods

Control Flow Analysis (Including Cyclomatic Complexity)

Control flow analysis can be conducted using tools or done manually at various levels of abstraction (module, node, etc.) and is done for the following reasons:

- Ensure the code is executed in the right sequence.
- Ensure the code is well structured.
- Locate any syntactically unreachable code.
- Highlight the parts of the code (e.g., loops) where termination needs to be considered.

This may result in diagrammatic and graphical representations of the code being produced.

Data Flow Analysis

The objective of data flow analysis is to show that no execution paths in the software exist that would access a variable not set to a value. Tools use the results of control flow analysis in conjunction with read/write access to variables. It can be a complex activity, as global variables can be accessed from anywhere. This analysis can also detect other code anomalies such as multiple writes without intervening reads.

Information Flow Analysis

Information flow analysis identifies how execution of a unit of code creates dependencies between the inputs to and outputs from that code. These dependencies can then be verified against the dependencies in the specification. This analysis is often particularly appropriate for a critical output that can be traced all the way back to the inputs of the hardware/software interface.

Information flow analysis may be augmented in some tools by using *annotations*. These are stylized comments that document certain assumptions about functions, variables, parameters, and types. They enable an analysis to proceed more efficiently because they give it more information relevant to a particular block of code.

Path Function Analysis (Also Called Semantic Analysis or Symbolic Execution)

Path function analysis is used to verify properties of a program by algebraic manipulation of the source text, without requiring a formal specification. It involves checking the semantics of each path through a program section or procedure. Sophisticated tools give expressions for the precise mathematical relationship between inputs and outputs from a particular program section: They effectively give the *transfer function* for that program section [12]. They step through the code, assigning expressions instead of values to each variable. Thus the sequential logic is converted into a set of parallel assignments in which output values are expressed in terms of input values – this format is easier to interpret. The tools produce an output for each path consisting of the conditions that cause the path to be executed, and the result of executing that path.

Semantic analysis reveals exactly what the code does in all circumstances for the

whole range of input variables for each program section. However, there is still the need for substantial human involvement in comparing the tool's output with the specification. Compliance analysis (formal code verification) provides a reduction in human requirements and greater automation.

Formal Verification (Also Called Compliance Analysis)

This is the process of proving, in an automated process and as far as is possible, that the program code is correct with respect to the formal specification of its requirements. All possible program executions are explored, which is not feasible by dynamic testing alone. Depending on the power of the tool being used, and its simplification ability, the involvement of analysts may be large or small.

Verification conditions can enhance compliance analysis. They consist of conditions that should be valid at the start and end of a block of code (pre- and post-conditions) and are stated at the start of that block. In a way, it is like a different form in which the programmer can explain the purpose of a block of code. The analysis might start with the post-condition and work backward to the start of the block. If, on reaching the start, the pre-condition is generated, then the block of code is provably sound.

Compliance analysis effectively performs a proof of the code against a low-level mathematical specification. In this respect, it is by far the most rigorous of the static analysis techniques. However, its value depends on the availability of a specification expressed in a suitable form. Furthermore, this rigor is at the expense of cost; productivity is around five lines of code per man-day.

Independent Evaluation

Since the 1970s, independent code inspections to reduce code error have been found to be efficient and cost effective. Experience from aircraft static code analysis carried out to date shows that code walkthrough finds about 60 percent of all the anomalies found.

Other Techniques

Syntax Checks

Syntax checks aim to find language rules violations such as using a variable of the wrong type or before it is declared. The compilers of some languages such as Ada and Pascal will carry out syntax checks automatically, whereas languages like C and assembler need additional tools.

To allow the use of analysis tools, reduce the number of likely coding violations and improve code readability. It is normal to define a rule set when designing safety-critical code to allow the tools to carry out the analysis more readily and to remove some of the more problematic features. It has been found that the size of the rule sets is dependent on the language, such as the following:

- C has some 220 rules suggested [13].
- Ada 83 has approximately 80 rules.
- Southampton Program Analysis and Development Environment (SPADE) Ada Kernel (SPARK Ada) has an extensively defined rule set; sometimes a reduction in the rules can be agreed on.

Range Checking

Range checking analysis aims to verify that the data values remain within their specified ranges, as well as maintain specified accuracy. This technique can detect the following:

> *"All software contains errors, and computer programs rarely work the first time. Usually, several rounds of rewriting, testing, and modifying are required before a working solution is produced."*

- Overflow and underflow analysis.
- Rounding errors.
- Array bounds.
- Stack usage analysis.

This is a form of shared resource analysis that establishes the maximum required size of the stack, and whether there is sufficient physical memory to support it.

Timing Analysis

Timing analysis ascertains the temporal properties of the input/output dependencies, including the worst-case execution time for the correct behavior of the overall system. It can be made difficult by language features such as manipulation of dynamic data structures, loops without static upper bounds, and by using hardware with built-in pipe and cache.

Other Memory Usage Analysis

This is required for any resource that is shared between different partitions of software. It reveals the absence of conflict between the code and other low-level components such as device drivers and resource managers.

Object Code Analysis

Object code analysis demonstrates that the object code is an accurate translation of the source code and that the compiler has introduced no errors. The analysis may be carried out by manual inspection of the machine code produced by the compiler – this can be made easier if the compiler vendor provides details of the mappings from source code to object code.

Limitations

Although the various forms of static code analysis offer many advantages to the system developer, they also impose some constraints. Using these techniques restricts language choices that may be used and the choice of the structures used within these languages. Furthermore, these analytical methods require highly skilled and experienced staff to carry out the tests and analyze the results. It is not a complete answer for the validation and verification of safety-critical software even with the use of automated tools. Other forms of testing (for example dynamic) are required to verify certain aspects, like executing critical features. Some of the restrictions of static analysis using automated tools are the following:

- Multitask applications software must be analyzed a task at a time. Another form of testing is required to check task interactions.
- Dynamic aspects of the software (for example sequences of program execution) are difficult to model with static analysis techniques.
- Most automated tools require translation to an intermediate language before they can analyze the code. Automatic translators are available for some languages, but for others one must either translate manually or write a new translator. Some language features do not have an equivalent in the intermediate language even with the automatic translators; they must be manually translated. The static analysis of the software depends on its translation model and the more skilled the analyst, the more skilled the model produced. The validation of the intermediate language model needs to be considered, as this can be a major problem.

Air Vehicle Software Analysis

The practical application of static code analysis has produced some interesting results. The range of software systems that have been subjected to analysis include the following:

- Automatic flight control.
- Engine control.
- Fuel and center-of-gravity management.
- Warning systems.
- Anti-icing systems.
- Flight management.
- Stores management.
- Air data units.
- Radio altimeters.
- Anti-skid brakes.

These systems vary in size from 3,000 lines of code to 300,000 lines of code and include languages from assembler, C, Pascal, Ada to Lucol, and SPARK Ada [14].

Effects of Previous Development Methodologies (RTCA DO-178B) [8]

It is worth reiterating that when comparing RTCA DO-178B [8] Levels A and B code, no discernible difference was found by static code analysis demonstrating that static code analysis is something you carry out in addition to testing. Even the most extensive testing does not remove the anomalies found by static code analysis. Surprising amounts of *dead* code have been found in code developed to RTCA DO-178B Levels A and B.

Effects of Language

The choice of language for a computer program is important. Not only should the functionality of the language itself be considered, but also the availability and quality of support tools and the expertise within the development team. Unfortunately, safety-critical software represents only a small subset of the global programming effort; most languages are not designed with high integrity requirements in mind. More commonly, commercial factors such as productivity and ease of use steer the development.

Some languages are better suited to the production of safety-critical software than others because they make it easier to write dependable code, and easier to demonstrate its freedom from errors. However, you must bear in mind that the language itself is a product that is susceptible to design flaws: Perfect code could still produce errors when run.

The software lines of code per anomaly in Table 1 show some of the metrics found from various programs. Table 1 shows that the poorest language for safety-critical applications is C with consistently high anomaly rates. The best language found is SPARK (Ada), which consistently achieves one anomaly per 250 software lines of code. The average number of safety-critical anomalies found is a small percentage of the overall anomalies found with about 1 percent identified as having safety implications. Automatically generated code was found to have considerably reduced syntactic and data flow errors.

Software development is often performed on a different system than that used for the final application. Therefore, the portability of the code is another factor to be considered when choosing a language. That is, how easily it will run in a different environment to the one in which it was developed.

The quality of the available compilers is also important. Modern programming languages are very complex and sophisticated and hence difficult to understand. It is therefore challenging to write high quality, dependable compilers for them [15]. Widely used compilers and development tools should be used whenever possible, so that there has been plenty of opportunity for errors to be found (and hopefully rectified). This also applies to the language used, reflecting why an attractive (but little used) language such as Modula-2 might not be chosen for a safety-critical application.

Table 1: *Software Language Anomaly Rates*

Software Language	Range	Software Lines of Code Per Anomaly	Anomalies Per Thousand Lines of Code
C	Worst	2	500
	Average	6 - 38	167 - 26
	Best (Auto Code Generated)	80	12.5
Pascal	Worst	6	167
	Average/Best	20	50
PLM	Average	50	20
Ada	Worst	20	50
	Average	40	25
	Best (Auto Code Generated)	210	4.8
Lucol	Average	80	12.5
SPARK	Average	250	4

Tools Available

There are a number of static code analysis tools available. They offer different depths of analysis, and some will only operate on a few languages. Most of them run on uncompiled source code and first translate to an intermediate language, which the analysis tool itself can read.

The time taken for the tool to analyze the code may be only a small fraction of the time taken to carry out static analysis of the code. Many tools produce reams of data that must be laboriously analyzed and processed; staff requires skill and a lot of training.

Main Tools

There are three main, well-established tools used on UK military programs.

Malvern Program Analysis Suite

Malvern Program Analysis Suite (MALPAS) was developed by Royal Signals and Radar Establishment Malvern based on research they carried out and by Southampton University in the 1970s. It is now mature and since 1986 has been supplied and supported by Advantage.

Although automatic translators exist for most languages, the main ones covered are Ada and Pascal. There is no concept of pointers in the MALPAS Intermediate Language and so to analyze C, for example, the code would first have to be purged of the use of pointers – a potentially formidable task.

SPARK

SPARK is a subset of Ada for high integrity programming first formalized by Bernard Carré and Trevor Jennings of Southampton University in 1988. It has continually evolved and nowadays it is being more widely used and is gaining general acceptance particularly as its tools now run within a *lunchtime* for an average-sized safety-critical avionics program. In addition, SPARK now supports tagged types, tasking (Ada95 Ravenscar), and *proof of exception freedom*, which have particular benefits in the context of RTCA DO-178B [8].

Control flow analysis is not needed as it is subsumed into the SPARK grammar, and thus performed *on the fly*. Data and information flow requirements have been expressed as SPARK program design rules.

Liverpool Data Research Associates Ltd. Testbed

Liverpool Data Research Associates Ltd. (LDRA) Testbed was founded in 1975 and is the oldest developer and retailer of static analysis tools. Many languages are covered: Ada, C, C++, Cobol, Coral 66, Fortran, Pascal, PL/1, PL/Mx86, and Intel and Motorola assemblers. The LDRA Testbed will perform the three flow analyses, with information flow analysis enhanced by the use of annotations.

Other Tools

Other tools include the following:

- SPADE.
- QA C Programming Research Ltd.
- Cantata and AdaTEST IPL [16].
- Alsys C-SMART – Certifiable Small Ada Run-Time [15].
- Aonix RAVENSCAR.
- PC-Lint [17].
- LCLint [18].
- PolySpace Technologies [19].
- Compaq Systems Research Center – Extended Static Checking (ESC) [20].

Conclusion

The safety argument approach should be used so that safety-critical software is minimized, safety functionality is clearly identified, and analysis required is justified.

Static code analysis is an effective software analysis technique; hence, its use is recommended in the context of safety-critical software particularly when conducted constructively as part of the software development process.

If it is conducted retrospectively, it is necessary to specify the nature and depth of any analysis carried out. Static analysis techniques should be targeted by the safety arguments. Techniques for targeted, rather than *blanket* analyses are being investigated by a number of organizations. Once developed, they may reduce the cost of analysis, while maintaining the required depth in the areas of interest.

Experience with retrospective static analysis shows that independent code walkthroughs are the most effective technique for software anomaly removal. These seem to find up to 60 percent of the errors present in the code.

The use of automatic code generation should be encouraged because this seems to result in low syntactic and data flow errors.

A safe subset of Ada must be considered when selecting a language for safety-critical systems, as this will ensure anomalies are minimized. SPARK continues to prove it is the most reliable approach to safety-critical software. However, C and its associated forms should be avoided.◆

References

1. Barnes, John. High Integrity Software – The SPARK Approach to Safety and Security. Addison-Wesley, 2003 <www.sparkada.com>.
2. Graham, Buckle. Static Analysis of Safety Critical Software. Proc. of the 16th Safety-Critical System Symposium, Springer, United Kingdom, 1998 <www.safety-club.org.uk>.
3. Wichmann, B. A., A. A. Canning, D. L. Clutterbuck, L. A. Winsbarrow, N. J. Ward, and D. W. R. Marsh. "Industrial Perspective on Static Analysis." Software Engineering Journal Mar. 1995: 69-75 <http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=8550>.
4. Ministry of Defense. "Design and Airworthiness Requirements for Service Aircraft." Part 1, Issue 2. Defense Standard 00-970 Dec. 1999 <www.dstan.mod.uk>.
5. Ministry of Defense. "General Specification for Aircraft Gas Turbine Engines." Issue 1. Defense Standard 00-971. 29 May 1987 <www.dstan.mod.uk>.
6. Ministry of Defense. "HAZOP Studies on Systems Containing Programmable Electronics." Part 1, Issue 2. Defense Standard 00-58. 19 May 2000 <www.dstan.mod.uk>.
7. Ministry of Defense. "Requirements for Safety Related Software in Defense Equipment." Part 1, Issue 2. Defense Standard 00-55. 1 Aug. 1997 <www.dstan.mod.uk>.
8. Radio Technical Commission for Aeronautics. "Software Considerations in Airborne Systems and Equipment Certification." RTCA DO-178B. RTCA, Inc., Dec. 1992 <www.rtca.org>.
9. Ministry of Defense. Regulation of the Airworthiness of Ministry of Defense Aircraft. 4th ed. JSP318b. Nov. 1999.
10. TA Consultancy Services Ltd. "MALPAS Training Course." TACS/9093/15. T A Consultancy, 12 Aug. 1992 <www.tagroup.co.uk/index.htm>.
11. Storey, Neil. Safety Critical Computer Systems. Addison Wesley Longman, 1995.
12. Proc. of the Advisory Group for Aerospace Research and Development Conference 545 <www.rta.nato.int/rtohistory/agard.htm>.
13. Hill, M., and L. Whiting. "Risk Reduction for C Coding." Internal QinetiQ Document. DERA Malvern, 1999.
14. Harrison, K. J. "Static Code Analysis on the C-130J Hercules Safety-Critical Software." Aerosystems International, UK, 1999 <www.damek.kth.se/RTC/SC3S/papers/Harrison. doc>.
15. Aonix <www.aonix.com>.
16. AdaTEST and Cantata <www.iplbath.com>.
17. Gimpel Software <www.gimpel.com>.
18. Networks and Mobile Systems <http://lclint.cs.virginia.edu/guide/index.html>.
19. Polyspace Technologies <www.polyspace.com>.
20. Compaq Extended Static Checking for Java <www.research.digital.com/SRC/esc/Esc.html>.

Note

1. This article is based on a paper that was presented to the Safety Critical Systems Club as "Air Vehicle Software Static Code Analysis Lessons Learned – Ninth Safety Critical Club Symposium," Bristol, United Kingdom: Springer, Feb. 2001, ISBN 1-85233-411-8.

Additional Reading

1. Ministry of Defense. "Safety Management Requirements for Defense Systems." Part 1, Issue 2. Defense Standard 00-56 Part 2 Issue 2. 13 Jan. 1996 <www.dstan.mod.uk >.
2. QAC <www.programmingresearch.com>.

About the Author

Andy German leads a group of software and safety engineers for the Test and Evaluation Sector of QinetiQ Ltd. This group provides the United Kingdom (UK) Ministry of Defense with independent certification advice for large military aircraft and Unmanned Air Vehicles. Andy has worked in the UK defense sea and air system sectors for the past 21 years and will complete a Master of Science degree in Safety Engineering from Lancaster University this year.

QinetiQ Ltd.
Safety and Signature Evaluation
Test and Evaluation Services
Room G10, Bldg. 498
MoD Boscombe Down
Salisbury, Wilts SP4 OJF
Phone: +44 (0) 1980 66 3987
Fax: +44 (0) 1980 66 3035
E-mail: agerman@qinetiq.com