Static Analysis Tools Pass the Quals

Yannick Moy, AdaCore

Abstract. Submitting a system to certification involves demonstrating, with a degree of confidence commensurate with the system's criticality, that it meets its requirements completely and correctly. The software life cycle process known as verification is responsible for achieving the relevant level of assurance and traditionally has relied on testing and manual reviews. Static analysis (SA) tools are starting to automate some of these verification activities. In this article we discuss what qualifies SA to be used as part of the software verification process in a certification context.

Certification and Qualification

Certification is the official process by which critical software is deemed fit-for-purpose. Depending on the domain, certification may be granted by an independent authority (civil avionics, railway, nuclear, space), the customer (military) or by an internal service within the company itself (automotive). Reference documents for certification may be international standards (civil avionics, railway, nuclear, space, automotive) or national standards (military).

In this article, we refer to the following documents:

- DO-178C (2011): civil avionics standard, known as ED-12C in Europe
- DO-333 (2011): formal methods supplement to DO-178C, known as ED-216 in Europe
- EN 50128 (2011): railway standard in Europe
- SWEHB (2013): NASA Software Engineering Handbook, guidance for implementing the NASA Software Engineering Requirements
- MISRA C (2012): coding standard for C, originally from automotive industry
- JPL C (2009): coding standard for C from NASA Jet Propulsion Laboratory

Qualification is the formal process by which software tools, for example static analysis (SA) tools, are deemed fit-for-purpose to address certification objectives. Tool vendors may provide a qualification kit that can be used by the applicant to qualify the tool in a specific certified project.

Soundness and Completeness

Soundness and completeness are usually used with different meanings in static analysis (SA) tools targeting bug finding and those targeting verification. In this article, we adopt the following definitions:

- A SA tool is sound if it never reports that a property is true when it may not be true. Reporting that an error might occur has no effect on soundness.
- A SA tool is complete if it reports all errors, either with certainty (the error occurs) or uncertainty (the error may occur).

Complete tools typically report uncertain errors where no actual error occurs. These are called false alarms or false positives. Incomplete tools typically fail to report actual errors. These omissions are called false negatives.

Static Analysis in Certification

There is no shortage of evidence that use of SA tools can increase code quality, to the point that a US military evaluator recently said that he would stop any project that did not use at least two of them. The phrase "at least two" points to repeated findings by the Static Analysis Tool Exposition (SATE) project from NIST, that different SA tools find different errors [1]. A corollary to that finding is that SA tools might not report errors in the categories that they handle. A striking example is the Heartbleed vulnerability in OpenSSL, a case of buffer overflow, which could not have been found by any SA tool before it was revealed publicly on 7th of April 2014 [2].

Most publications on SA, including previous publications in CrossTalk [3,4,5], focus on the use of SA for code quality, where the criterion of reporting only the most probable and understandable errors is justified. This is in contrast with the use of SA tools for addressing certification objectives, where the lack of soundness and completeness is inappropriate. While this use of SA is not new [6], we have seen an increasing industrial adoption in the last few years, as a cost-effective alternative to reviews or testing. In this article, we discuss what processes are used in practice to adopt SA tools in certification, based on our experience in building qualifiable SA tools at AdaCore: the coding standard checker GNATcheck, the static analyzer CodePeer, and the formal verification SPARK toolset.

As the process varies greatly depending on the category of tool, we consider separately the three categories corresponding to these tools: coding standard checking, detection of programming errors, compliance of implementation with design. These three categories are representative of the types of SA tool capabilities that are used in certification. Before considering these categories, we discuss some general aspects of SA.

Static Analysis of Source Code

Certification is concerned with getting high levels of confidence that the software behaves as it should and doesn't have undesirable behavior. SA tools used to automate verification should be sound, as expressed in DO-333: "The soundness of each formal analysis method should be justified. A sound method never asserts that a property is true when it may not be true." For SA to replace, not just complement, other activities, it should also be complete, as expressed in SWEHB: "For critical code, it is essential to use sound and complete static analyzers. Sound and complete analyzers guarantee that all errors are flagged and that no false negatives (i.e., an erroneous operation being classified as safe) are generated." This implies restrictions on programs or properties, which we discuss later, since it is impossible for any tool to be sound and complete for all conceivable properties and totally unrestricted programs.

It is important to realize that SA tools that analyze source code only analyze a model of the real executable program. Unless a programming language was primarily designed for SA (like B Method [7] or SPARK [8]), it almost certainly contains ambiguous constructs, which may be interpreted differently by a SA tool and by a compiler. In imperative programming languages such as Ada, C, C++ or Java, there are three classes of ambiguous constructs:

- Implementation-defined behavior. It is specified, possibly differently, by each compiler, for example the size of standard scalar types.
- Unspecified behavior. The set of possible effects of a construct is defined by the language but is not necessarily specified by the implementation's documentation, for example the order of evaluation in expressions.
- Undefined behavior. The effect of program execution is unpredictable, for example reading an uninitialized variable.

These can be addressed at three levels: the language, the execution platform and the compiler.

Addressing the Language Issue

For many cases of undefined behavior, the default language rules do not allow automatically separating correct from erroneous programs. For example, in C it is undefined to assign twice to the same variable inside the same sub-expression: "Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression." In the presence of calls and aliasing, no available SA tool can currently perform this check on unrestricted C programs. Instead, they can check more restrictive rules that prevent this type of undefined behavior, for example rule 19 of JPL C, which states "Do not use expressions with side effects", or rules 13.2, 13.3 and 13.4 of MISRA C, which restrict assignment operations.

Although it is conceivable to mirror all possible compiler-specified behavior in an SA tool, it is impractical because of multiple compilers with many version and switch variants. Instead, language rules (or coding standards) are defined that prevent this type of unspecified behavior. For example, the order of evaluation of parameters in a function call are not specified in Ada, C or C++, which can lead to different observable behaviors if expressions have side effects. By completely prohibiting such side effects, the rules defining the SPARK subset of Ada or rule 19 of JPL C ensure that all orders of evaluation will be equivalent.

Addressing the Execution Platform Issue

Many cases of implementation-defined behavior depend on the execution platform, for example the size of standard scalar types or the behavior of calls to run-time functions. This issue can be partially resolved through parameters to the SA tool.

Addressing the Compiler Issue

In those cases of unspecified and implementation-defined behaviors that are neither prevented by language rules nor resolved by the choice of execution platform, the SA tool needs to be able to handle all possible compiler choices. This is easier to achieve if the SA tool shares code with the compiler used to generate the executable program. This is for example the choice that we have made at AdaCore, when reengineering the CodePeer and SPARK SA tools by basing them on the GNAT Ada frontend for the GCC compiler. [9]

Coding standard checkers are probably the most widely used SA tools in certification. They automate the verification of language restrictions such as those mentioned earlier.

Coding Standard Checking

All certification standards require the definition of a coding standard, defining a set of rules that specify which language features are allowed (or prohibited). Typically, a coding standard contains rules that forbid some language features, and rules that limit code size/coupling/depth. Well-known coding standards are MISRA C [10] in automotive or the JPL C [11] at NASA.

The first objective of a coding standard is to prevent the introduction of certain types of programming errors. As SWEHB says: "The use of uniform software coding methods, standards and/or criteria ensures uniform coding practices, reduces errors through safe language subsets, and improves code readability."

The second objective of a coding standard is to facilitate detection of programming errors by testing and SA. For example, EN 50128 defines an objective "to identify programming constructs which are either error-prone or difficult to analyze, for example, using static analysis methods", with the aim that "a language subset is then defined which excludes these constructs." Similarly in DO-178C, an objective "is to ensure the Source Code does not contain statements and structures that cannot be verified". Indeed, SA tools are not sound on the entire language they target, for example when expressions have side effects or when arbitrary pointer conversions are allowed. Thus, a subset of the language should be defined in a coding standard, such that sound SA is possible. For Ada, an ISO technical report defines the compatibility between language features in Ada and classes of SA. [12] SPARK is an example of such a subset of Ada, which allows sound data-flow, information-flow, robustness and functional analyses.

Rules may be verified automatically, manually, or by a combination of both. Experience shows that manual verification, in particular when outsourced and performed only once on the final source code to decrease costs, is both error-prone and time consuming. Thus, most rules are usually verifiable automatically. For example, MISRA C distinguishes between decidable guidelines (116), where a guideline is decidable "if it possible for a program to answer with a yes or a no in every case", and undecidable ones (43). As another example, JPL C (120 rules) was applied to develop the code of the Curiosity rover, which landed on Mars in 2012, and "compliance with all risk-based rules could therefore be verified automatically with multiple independent tools on every build". [13] Only automatic tools can deal with the large number of rules in these coding standards.

As the previous quote indicates, automation also makes it possible to check the coding standard as part of the development process, catching violations as soon as they occur. Even better, most rules only require analysis of a relatively few surrounding lines of code and can therefore be analyzed very quickly on a developer's machine. There is much to gain in adapting the process to facilitate frequent analysis:

- All rule violations should be justified so that running the SA tool returns without finding any unjustified violations.
- The SA tool should be integrated in the developer's IDEs, so that it is as easy to run as it is to compile the code.
- The SA tool should be integrated in the version control system (using features for pre-commit checks) or build system. This ensures that one developer's error gets detected before it impacts others.

With the correct set of automatable rules, SA tools, and developer-centric processes, coding standard checking can be applied continuously. As SWEHB puts it: "Assuring the adherence of the developed software to the coding standards provides the greatest benefit when followed from software development inception to completion." Maximizing automation makes it possible to invest user efforts in the richer types of SA that follow.

Detection of Programming Errors

While coding standard checking is mostly concerned with decidable rules, detection of programming errors is mostly concerned with undecidable properties. Thus, a complete SA will necessarily report false alarms (possibly many), which need to be addressed in the process.

The usual main categories of programming errors detected by SA are run-time errors (buffer overflow, integer overflow, read of uninitialized data, null pointer dereference, etc.), dead code, concurrency errors (race condition, deadlock), and operating system and SQL injection vulnerabilities.

Run-time errors correspond to improper behavior according to programming language rules: a predefined exception is raised, or the behavior becomes undefined. Because there is a precise definition of where these errors occur, a complete SA is possible. But as there are many such possible errors in a program, there may be also many false alarms.

There are variations among domains on exactly which kinds of dead code, which may also be called extraneous code or unreachable code, are relevant. The two main kinds are code "which exists as a result of a software development error but cannot be executed" (DO-178C) or code "that is executed but whose removal would not affect program behavior" (MISRA C). Both are equally impossible to detect completely by SA. Instead, SA tools only report unconditionally unreachable code, with no false alarms.

If the detection of errors is incomplete, as with dead code detection, SA cannot replace testing or review activities, but can only complement them. If the detection of errors is sound and complete, which is possible for run-time error detection, SA can lower the effort on testing or review activities. Whether this can be done in practice for a given project, with a given tool, depends on the confidence that the team has in the tool. When the tool has been successfully used in practice over a long period of time, service history may provide enough confidence. Otherwise, tool qualification should be performed. Firstly, the SA tool should be used on a subset of the language on which it is sound and complete. Secondly, all unsound or incomplete heuristics meant to minimize the number of false positives in bug-finding mode should be deactivated. In our work on SA tools at AdaCore, both the language subset and the deactivating switches form part of the qualification kit that we ship to our customers. They define the qualified interface ensuring sound and complete analysis. Both subsets also ensure that tool qualification is feasible by limiting the cost of developing the tests that are needed for qualification.

The process varies depending on the time needed to run the tool, and on the precision of results:

1. The ideal situation is that developers can run the tool frequently, with few false alarms. This is typical of modular analyses. Developers are expected to commit code that passes the SA without errors, with all false alarms justified.
2. In a more common setting, the tool takes too long to be run by developers, but there are still few false alarms. Instead, developers have access to the results of a nightly run. This is typical of the global analysis performed in abstract interpretation-based SA, paired with a restricted language subset. Developers or some designated team members are expected to report/fix errors and justify false alarms on a daily basis.
3. In probably the most common setting, the run is long and produces too many false alarms to be integrated in the daily development process. This is typical of global analysis of programs with pointers, dynamic allocation, indirect calls, etc. Manual review of the analysis results is performed only at planned milestones, either by the development team or a dedicated Quality Assurance team.

While detection of programming errors mostly requires post-hoc manual work for dealing with false alarms, richer types of SA such as compliance of implementation with design require more preliminary work.

Compliance of Implementation with Design

Just as important as the absence of programming errors, the source code should implement the software design. In DO-178C, source code should be shown to be compliant with the two parts of design that are software architecture and low-level requirements (a.k.a. function specifications). Some specification languages make it possible to describe architectural constraints and low-level requirements, which allows developers to demonstrate by SA that the implementation satisfies those constraints and requirements.

This is illustrated in the languages of annotations for C programs used at Airbus in conjunction with SA tools Caveat and Frama-C [14], as described in DO-333 under the name of unit proof. First, low-level requirements are expressed as function contracts, consisting in a precondition (on entry to the function, established by the caller) and a postcondition (on exit from the function, established by the function itself). Then, code is developed to implement those requirements. Finally, SA is used to prove that the implementation satisfies the requirements. The SPARK subset of Ada similarly allows checking low-level requirements expressed as subprogram contracts, as well as architectural constraints expressed as data flows or constraints on calls. These types of SA were used on the UK military programs SHOLIS and C130J Hercules (submitted to DEFSTAN 0055 certification). [8] The B Method used in railway also allows proving high-level software requirements, at the cost of more manual proof effort. [7]

The process to apply SA depends here on the complexity of the properties to prove. Simpler properties like absence of run-time errors (see previous section) are mostly proved automatically, with the occasional addition of simple preconditions and postconditions. In our experience with SPARK, more complex properties require some tuning:

1. If contracts are executable (as in SPARK 2014 or E-ACSL contracts in Frama-C), run tests to detect most cases of mismatch between contracts and code.

2. Interact with automatic provers through the IDE to correct code or contracts and ensure the property is proved automatically.

3. If a property cannot be not proved automatically, either default to testing if the integration of proofs and tests is possible, or turn to manual proof in a theorem prover, or add a manual justification.

The key feature of these SA tools is that they analyze each function in turn, using user-written function contracts to analyze calls. This type of highly modular SA can be run easily in parallel or on a distributed farm of servers, making it possible to integrate in the daily development process.

Conclusion

We have described the processes that we have seen applied in practice to use three categories of SA tools in certification. Coding standard checking is a "must-have" of every project in certification. Detection of programming errors is a best practice, recognized for both reducing costs and improving quality. Compliance of implementation with design has been embraced by pioneers like Airbus [14, 15]. In all three categories, use of sound and complete SA tools guarantees that no error has been missed, which is essential when looking for the last error. Of course, this depends on there being a sound and complete SA tool for the property of interest and the programming language subset used.

Acknowledgements

This article benefited greatly from comments of colleagues at AdaCore, as well as reviews from Paul Black, David Mentré and Franck Sadmi, which I'd like to thank here.

ABOUT THE AUTHOR

Yannick Moy is a Senior Software Engineer at AdaCore, where he works on software source code analyzers, mostly to detect bugs or verify safety/security properties. Yannick previously worked on source analyzers for PolySpace Technologies, France Telecom R&D and Microsoft Research. Yannick holds an engineering degree from the Ecole Polytechnique, an MSc from Stanford University and a PhD from Université Paris-Sud. He is a Siebel Scholar.

AdaCore
46 Rue d'Amsterdam
Paris, France 75009
Phone: +33.1.4970.6718
Fax: +33.1.4970.0552
E-mail: moy@adacore.com

REFERENCES

1. Paul E. Black, "Static Analyzers: Seat Belts for Your Code", IEEE Security & Privacy, 2012
2. David A. Wheeler, "How to Prevent the next Heartbleed", 2014. <http://www.dwheeler.com/essays/heartbleed.html>
3. Paul E. Black, "Static Analyzers in Software Engineering", CrossTalk, March-April 2009
4. Yannick Moy, "Static Analysis Is Not Just for Finding Bugs", CrossTalk, Sep-Oct 2010
5. Piyush Jain, Ramakrishna Rao and Sathyanand Balan, "Challenges in Deploying Static Analysis Tools", CrossTalk, July-Aug 2011
6. Andy German, "Software Static Code Analysis Lessons Learned", CrossTalk, Nov 2003
7. Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier, "Météor: A successful application of B in a large project", FM'99
8. Ian O'Neill, "SPARK - A Language and Tool-Set for High-Integrity Software Development", chapter in "Industrial Use of Formal Methods: Formal Verification", Wiley, 2012
9. Johannes Kanig, Edmond Schonberg and Claire Dross, "Hi-Lite: the Convergence of Compiler Technology and Program Verification", HILT'12
10. MISRA, "MISRA C:2012 Guidelines for the use of the C language in critical systems". <http://www.misra-c.com/>
11. NASA Jet Propulsion Laboratory, "JPL Coding Standard for Flight Software". <http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf>
12. ISO/IEC, "Guide for the Use of the Ada Programming Language in High Integrity Systems", 1999 <http://www.open-std.org/jtc1/sc22/wg9/n359.pdf>
13. Gerard J. Holzmann, "Mars Code", Communications of the ACM, Feb 2014
14. Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels and Benjamin Monate, "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience", IEEE Software, 2013
15. Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré and Yannick Moy, "Rail, Space, Security: Three Case Studies for SPARK 2014", ERTS2, 2014