Static Analysis Is Not Just for Finding Bugs

Dr. Yannick Moy
AdaCore

Static analysis tools are gaining popularity for safeguarding against the most common causes of errors in software. The main focus of these tools is on automatic bug-finding—the first stage in a two-phase process where the tool finds bugs and the human then corrects them. This article explains that such a goal is too narrow for critical software assurance (SwA). Instead, static analysis tools should adopt a broader perspective: computing properties of software.

Static analysis tools (see the sidebar on page 7) are very useful for finding bugs. They go far beyond the capabilities of compilers (warnings) and coding standard checkers to which they are directly related. Like compilers when they generate warnings, static analysis tools aim to detect possible run-time errors (e.g., buffer overflow) and logic errors (e.g., variables not referenced after being assigned). Like coding standard checkers, static analysis tools sometimes allow users to define their own set of patterns to flag. But static analysis tools generally perform much more sophisticated analyses than is typically found in compilers and coding standard checkers (e.g., looking at global context and keeping track of data and control flow).

The appeal of these tools is immediate, providing an almost *yes/no* answer to very hard problems (termed *undecidable* in mathematical terms). But while you're asking *Are there any bugs in this code?*, the tool is actually answering a subtly different question: *Have any bugs been detected in this code?* Thus, when a tool answers *no problems*, it means that it couldn't detect any bugs; it doesn't mean that the code has no bugs. Further, the actual question should be: *Have any shallow/common bugs been detected in this code?* As explained by a team at software integrity company Coverity: "errors found with little analysis are often better" because they are clear errors that a human reviewer will more likely understand [1].

That is the catch. Static analysis tools are not compilers whose output (object code) rarely needs to be inspected. They produce results for humans to review. At the very least, a human needs to understand the problem being reported—and also, in most cases, the reason for the report—in order to assess what, if any, correction to make.

Focusing on Human-Readable Output

Because humans ultimately label each problem reported by a static analyzer as either a *real error* or a *false alarm* whose ratio is used to evaluate the quality of a tool, commercial tools strive to present the user with understandable warnings supported by explanations. Even trivial changes to the messages may have a large impact. In my own experience working on the static analyzer PolySpace, I was quite surprised by the positive response from customers on what seemed to be simply a cosmetic change. Messages for warnings had been reworded to reflect the associated likelihood, so that the message *out of bounds array index* associated with certain (red) and possible (orange) warnings was now turned into *Error: array index is out of bounds* and *Warning: array index may be out of bounds*.

The short message is usually accompanied by a link to a page describing the intent of the checker being exercised, and the typical errors that it finds. Some static analyzers also display more contextual information that helps the user in diagnosing the problem. For example, PREfix, an internal tool at Microsoft, displays whether the problem occurs inside a loop or not, the depth of calls that exhibit the problematic execution, etc.

As most problems only show up in some executions reaching a particular program point, a useful piece of information is the execution path leading to these problematic executions. Static analyzers typically display such paths by coloring the lines of code defining the path (e.g., the first line of each block of code involved). The path may involve function calls, in which case the user can usually unfold the call to follow the path. Some static analyzers even display contextual explanations along the path to help follow the rationale for a given warning.

Still, as Coverity's team puts it, "explaining errors is often more difficult than finding them" [1]. This means that a balance is found in practice between explaining displayed warnings and hiding those warnings that cannot be so easily explained. As a result, real errors—which are detected but are complex to explain—may fail to be reported: "For many years we gave up on checkers that flagged concurrency errors; while finding such errors was not too difficult, explaining them to many users was" [1].

Static Analysis for Critical SwA

Finding bugs with static analysis tools, even simple bugs that testing would catch is, of course, useful. Embedded systems expert Jack G. Ganssle advocates doing inspections before testing because inspections are 20 times cheaper than writing tests: "It is just a waste of company resources to test first" [2]. As human time is far more expensive than CPU time, the same argument shows that static analysis should be performed before inspections or testing, even for finding simple bugs.

However, the nets that static analysis tools are using to catch bugs have a large mesh, too coarse for critical SwA. One example is integer overflow: adding two large positive integers and getting a negative integer as a result. These are rather unimportant bugs for most commercial static analyzers, and are usually not even advertised on the list of vulnerabilities they look for. There is some rationale as to why integer overflow is not a high priority. At Microsoft Research, I worked with a team that augmented PREfix with the ability to detect integer overflow bugs—and then applied it to a large Microsoft codebase comprising several million lines of C and C++ [3]. The tool returned with tens of thousands of possible integer overflows—and almost all of them were intended or benign. With special heuristics to hide most false alarms, the tool returned with many fewer warnings (still hundreds). Three days of reviewing warnings finally uncovered 15 serious bugs, most of which were related to security issues. Relying on user review to find a few serious bugs amidst a large number of warnings is not the image that commercial static analyzers are trying to achieve.

Static Analysis for Automated Code Review

Instead of advocating a fully-automated approach that considers human review as a bottleneck in the application of static analysis, some have taken the opposite view and regard static analysis as a mechanism that can expedite manual code review.

Brian Chess and Jacob West from Fortify Software devote a complete chapter in [4] to static analysis as part of the code review process. They consider warnings issued by static analysis tools as clues that a non-trivial safety or security argument has to be made by a human reviewer, based on the fact that "static analysis tools often report a problem when they become confused in the vicinity of a sensitive operation" [4]. They also insist that, whenever possible, a problem found by code review that is not reported by the tool should be the basis for a new custom rule in the static analyzer. Although many tools supply an application programming interface for defining such custom rules, it is not likely that most errors found during code review can be easily encoded into such rules (keep in mind that several organizations can create custom checkers).

Tucker Taft and Robert Dewar have gone further, explaining how to leverage static analysis tools for automated code review [5]. This requires a way to query the internal information computed by the tool instead of just the warnings it issues. They show how to conduct a code review of inputs and outputs, preconditions and postconditions, etc., based on information generated by static analyzer CodePeer. Undoubtedly, making static analysis a partner in code review presents many questions concerning the interaction between the tool and the reviewer: One must determine how much information to display, how to display it, and which queries should have displayed the information. So far, static analysis tools have largely stayed away from this issue because of the difficulties in dealing with the large amount of information available.

However, combining static analysis with code review holds the promise of each method complementing the other, since their strengths are in different areas. Tools are deterministically sound and unsound (whether by design or through errors in the tool itself or in its setup), while humans are unpredictably sound and unsound. I recently co-conducted a very small experiment to compare the results of static analysis and code review for finding bugs in a Tokeneer system [6]

whose security properties were formally verified. The results of this experiment suggest that each method catches bugs the other method misses.

Focus on Humans, Not on Bugs

Orienting static analysis towards automation-assisted code review requires shifting the focus from finding bugs to helping a human understand various issues about the code, from data-flow to exception handling to proper input validation. This does not mean abandoning warnings. On one hand, tools are very good at systematically detecting a clearly defined problem, whereas humans make errors. On the other hand, tools cannot easily deal with the specific project issues or translate informal specifications into code verification activities. Michael D. Ernst believes that "humans are remarkably resilient to partially incorrect information, and are not hindered by its presence among (a sufficient quantity of) valuable information" [7].

The idea is to automate all the things that can be automated, but no more. With enough eyes, all bugs are shallow. We cannot say the same about enough tools. The choice of what is and is not important is best left to a human to decide, provided suitable user interactions are built into the tools. The problem is that static analyzers targeted at bug-finding may not be so easy to re-architect for answering queries from a user. Many of these tools only consider sets of execution paths that do not cover all cases; therefore, they may not easily provide information on all executions.

Tools like PolySpace and Frama-C display ranges of integer variables (and pointer variables for Frama-C) on demand: When the user puts the focus on a variable in the code, the range corresponding to all the possible values of this variable (in all executions) is displayed in a tool-tip or in a side panel. PolySpace uses the same kind of interaction to display all the information it computes about possible run-time errors; it is emphasized by coloring the code using the standard convention of green for *ok*, orange for *warning*, and red for *error*.

Static Analysis for Computing Properties

An absence of run-time errors is the first property that comes to mind when talking about static analyzers. Most tools cannot compute this property, as they are designed to report only a subset of all possible errors and analyze only a subset

of all possible executions. To the best of my knowledge, only three commercial tools compute this property: the Poly-Space and CodePeer tools, and the SPARK programming language. By focusing on humans rather than bugs, all three have found ways to solve the *false alarm* problem: PolySpace colors the code and lets users query individual program points for possible run-time errors; CodePeer partitions warnings into three buckets (high, medium, low) with low warnings only presented on user request; and SPARK imposes enough restrictions (checked by static analysis) that the false alarm rate is low (e.g., there can be no read of an uninitialized variable). All of these tools also allow recording manual analysis of warnings for reuse when the code is reanalyzed after being modified.

Absence of run-time errors is not the only property of interest in critical SwA. In fact, it is rather the least interesting property (things behave as they are written), except that it must hold in order for the program to respect any other property, and it could ideally be verified from source code only without any user guidance. Absence of run-time errors is sometimes framed as program correctness, which tends to boost its importance.

In a recent position paper [8], software engineering pioneer David Lorge Parnas warns that this abstract notion of correctness makes no sense in practice: "Correctness is not the issue." Indeed, correctness is always relative to a given specification and every non-trivial specification is wrong, whether it is formal or informal. The usual *wrongness* is being incomplete. This is especially true for formal specifications, because no existing formal language can express all the properties we expect from a correctly operating system, in particular for embedded software that interacts with the outside world. As an example, a correct compiler is one that must satisfy a number of requirements, including the issuing of useful error messages. No formal language can express this specification. Instead of correctness proofs, Parnas urges static analysis tool writers to focus on property calculation, which is the norm in other engineering fields.

We are interested in two types of properties:
- Functional properties like values, relations, preconditions, postconditions, and dependencies.
- Non-functional properties like coverage, memory footprint, worst-case execution time (WCET), and profiling.

Most static analyzers are already capable

of generating functional information because they internally compute program invariants that are predicates describing some constraints respected by the program (e.g., the fact that variable X is positive at some point, or more complex relations between variables like linear inequalities and Boolean combinations of such inequalities that hold at some point). Preconditions and postconditions are special kinds of invariants that are particularly interesting, because they make function interfaces explicit.

The first problem is that a static analyzer may compute a large number of such invariants, most of which are not of interest to the user. As already mentioned, one solution is to let the user indicate which invariants are of interest. Some tools already display the ranges of values taken by variables when a user selects such a variable in the program. Ideally, we would like to provide an arbitrary expression, say $X + Y$, and ask the static analyzer for all invariants at a specific program point that mentions this expression.

A second problem is that many static analyzers do not exactly compute invariants, either because they analyze only one path (or set of paths) at a time, or because they perform unsound simplifications during their analysis. In the former case, the predicate that characterizes the path (or the set of paths) analyzed is usually not easily readable, so simply outputting invariants of the form *predicate-for-the-path* implies *invariant-for-the-path* is unlikely to be useful. Instead, we can imagine that the path (or the set of paths) is displayed by highlighting appropriate lines in the source code (as is already done for warnings)—and that only the invariant part is displayed. Even in the case where the static analyzer performs unsound simplifications (possibly missing a real error), giving access to the internal invariants may help users understand the simplifications performed by the tool. When looking for integer overflow bugs in a large codebase at Microsoft, I found it very useful to have access to the models computed by PREfix for each function. These models gave the invariants at function exit (postconditions) computed by the tool for a set of paths described by invariants at function entry (preconditions). This was critical to quickly discard warnings caused by an incorrect model computed by the tool, which made it possible to concentrate on actual errors.

Some static analyzers also compute non-functional properties (i.e., properties that are not related to the correctness of the program's computations). Many static analyzers warn about dead code, which is

---

What Is a Static Analysis Tool?

A static analysis tool (or static analyzer) has three major characteristics:
- Its input is the source code for a program in a programming language.
- It analyzes the program's structure without executing the program.
- As its primary function, the tool outputs information that is relevant to humans developing or maintaining the program.

This general definition includes tools such as coding standard checkers, *bug finders*, and test case generators.

Many static analysis tools attempt to detect problematic constructs. Ideally, such a tool should identify all constructs in a given program—and only those constructs encountering the problem during execution. Unfortunately, mathematical computability theory shows that it is impossible to produce such a tool for analyzing arbitrary programs in any nontrivial programming language. So, in practice, a tool will suffer from either one or both of these deficiencies:
- Failure to detect a problem, yielding what is (perhaps confusingly) known as a *false negative*.
- Mistakenly flagging a correct construct as a problem, yielding a false alarm (known in the literature as a *false positive*).

A tool that does not generate false negatives is said to be *sound*. A tool's precision is a measure of its ability to avoid generating false positives. Soundness and precision are tradeoffs, so the challenge for a tool provider is to strike an appropriate balance.

---

the same property as code coverage, only seen from the other direction. Although general coverage seems hard to attain by static analysis, unit coverage that considers the coverage of a function's constructs for all possible calling contexts (and thus all values of inputs) is much more feasible. Again, mapping the results of the analysis onto the source code provides the best user interaction here. Generating tests whose execution shows a line of code is also a constructive way to compute the property that a line of code is not dead.

Expanding on this idea, we can imagine giving a predicate at a program point, say $X < Y$, and asking the static analysis tool to produce a counterexample. This is a very efficient way to make progress when the tool does not generate an invariant which, according to the user, should hold. Without such interactions, the user is usually left wondering if the tool was not clever enough to prove the property—or if it holds at all. Additionally, seeing the actual counterexample (instead of only knowing there is one) greatly facilitates understanding of the problem. What is important here is the user interaction, which allows very quick feedback on a question that the user finds interesting.

Specialized static analysis tools already provide information such as memory footprints and WCET. For example, Airbus is using these tools to help certify their programs at the highest levels of the DO-178 avionics safety standard [9]. However, not much work has been done with these tools to provide a rich user interaction at the function level.

New ways of interacting with static

analysis tools are desirable and possible. As a very simple example, some integrated development environments (IDEs) can display the shortest path in the call graph between two functions when a user asks whether one can be called from the other. Other IDEs highlight entities based on syntactic categories, triggered when the user puts the cursor on an entity. Those are the kinds of useful interactions that static analyzers should aim for.

Conclusion

The current emphasis on static analysis will not necessarily provide the tools that are needed for critical SwA, which is based on human assessment of *fitness-for-purpose*. Useful tools are those that compute human-readable properties of the software, providing reviewers with much deeper information than is currently available. The Agile Manifesto [10] correctly recognizes that individuals and interactions should be our main focus for creating useful processes and tools.

One static analysis vendor goes as far as to admit: "No one wants to be on the hot seat when a critical vulnerability is exploited in the field or when a coding mistake causes product recalls, brand damage, or revenue losses." I do not think that static analysis provides the kind of insurance suggested in [11]; like other systems assurance, critical SwA is not principally a matter of tools, but a matter of "leadership, independence, people, and simplicity" [12].

Static analysis for code review is certainly a very promising venue for critical SwA. Looking even further, static analysis

Software Defense Application

The defense industry—as evidenced by projects such as Software Assurance Metrics and Tool Evaluation—is paying significant attention to static analysis tools. This article helps DoD decision-makers and developers assess and select static analysis tools that meet their safety and security requirements.

used during development (e.g., for code review preparation) can help a programmer understand complex behaviors and detect subtle mistakes—like a "buddy" does in pair programming. In other words, static analysis for humans.◆

Acknowledgements

Many colleagues at AdaCore provided very valuable comments on an initial version of this article, in particular Bob Duff and Ben Brosgol.

References

1. Bessey, Al, et al. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World." *Communications of the ACM* 53.2. Feb. 2010 <http://cacm.acm.org/magazines /2010/2/69354-a-few-billion-lines-of -code-later >.
2. Ganssle, Jack G. "A Guide to Code Inspections." Vers. 2.1. Feb. 2010 <www.ganssle.com/inspections.pdf>.
3. Moy, Yannick, Nikolaj Bjorner, and Dave Sielaff. "Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis." *Microsoft Research*. 11 May 2009 <http:// research.microsoft.com/apps/pubs/ ?id=80722>.
4. Chess, Brian, and Jacob West. *Secure Programming with Static Analysis*. Chapter 3, "Static Analysis as Part of the Code Review Process." Upper Saddle River, NJ: Addison-Wesley, 2007 <http:// media.techtarget.com/searchSoftware Quality/downloads/Secure_Program ming_CH03Chess.pdf>.
5. Taft, S. Tucker, and Robert B.K. Dewar. "Making static analysis a part of code review." *Embedded Computing Design*. 16 June 2009 <http://embedded-compu ting.com/making-static-analysis-part -code-review>.
6. Moy, Yannick, and Angela Wallenburg. *Tokeneer: Beyond Formal Program Verification*. Proc. of the Embedded Real Time Software and Systems Conference. Toulouse, France. 21 June 2010 <www.open-do.org/wp-content/up loads/2010/05/erts2010.pdf>.
7. Ernst, Michael D. *Static and Dynamic Analysis: Synergy and Duality*. Proc. of the Workshop on Dynamic Analysis. Portland, OR. 9 May 2003 <www. cs.washington.edu/homes/mernst/ pubs/staticdynamic-woda2003.pdf>.
8. Parnas, David Lorge. "Really Rethinking Formal Methods." *IEEE Computer* 43.1 (Jan. 2010).
9. Souyris, Jean, et al. *Formal Verification of Avionics Software Products*. Proc. of the 16th Annual Symposium on Formal Methods. Eindhoven, The Netherlands. 2-6 Nov. 2009.
10. Beck, Kent, et al. "Manifesto for Agile Software Development." Feb. 2001 <www.agilemanifesto.org>.
11. Fisher, Gwyn. "When, Why and How to Leverage Source Code Analysis." White Paper. 2007 <www.klocwork. com/resources/white-paper/static -analysis-when-why-how>.
12. Haddon-Cave, Charles. *The Nimrod Review: An Independent Review into the Broader Issues Surrounding the Loss of the RAF Nimrod MR2 Aircraft XV230 in Afghanistan in 2006: Report*. London: TSO. 28 Oct. 2009 <http://ethics. tamu.edu/guest/XV230/1025%5B 1%5D.pdf>.

About the Author

Yannick Moy, Ph.D., is a senior software engineer at AdaCore, where he works on software source code analyzers CodePeer and SPARK, mostly to detect bugs or verify safety/security properties. Moy previously worked on source analyzers for PolySpace (now The MathWorks), INRIA Research Labs, Orange Labs, and Microsoft Research. He holds degrees in computer science: a doctorate from Université de Paris-Sud, a master's from Stanford, and a bachelor's from the Ecole Polytechnique. Moy is also a Siebel Scholar.

AdaCore
46 Rue d'Amsterdam
Paris, France 75009
Phone: +33.1.4970.6716
E-mail: moy@adacore.com