**Static Code Analysis**

**Tim Biggin**

**Abstract**

Static code analysis may not be a new software concept; modern tools offer helpful techniques to facilitate error detection. Through the creation of new coding standards and methods of detection, these checkers are proving to be vital tools in the fight against illusive software bugs and run-time errors. Such checkers can be used long before other methods of bug locating, such as creating test cases or debugging code. Although these tools do have their downsides, integrating them into the software development life cycle can save time and money in the long run. These tools can assist in instructing new employees within organizations as well as enforcing formatting. There exist a variety of tools, with which you will find the right combination for your uses, but only through experimenting and comparison.

**What Static Code Analysis Does**

Static code analysis is defined as a method of detecting errors and defects located in the source code of a program. A static code analyzer can thoroughly analyze code and make suggestions where the code should be changed or modified based on rules defined by the user. The range of errors that a static analyzer can detect is quite diverse, ranging from coding defects to meeting coding standards such as MIRSA-C/C++ or CERT C/C++. Such tools can also help with formatting the style (indents, spaces, tabs, etc.) of code by ascertaining if written code follows the style used by a company or organization. Static analyzers can also produce various code metrics, such as lines of code, in thousands (KLoC); file counts and “churn”; which is the number of files changed between builds [7] [12]. These metrics can be used to indicate the quality of the analyzed code. More advanced tools, such as Polyspace by MathWorks, offer more complicated features. Such tools implement a formal methods-based approach using mathematical proofs to confirm the absence of run-time errors [8] [1] [6].

These tools are entirely automated and analyze 100% of the source code without compilation, execution, or the use of test cases. Execution paths are analyzed by the tool, and variable ranges and concurrent data access points are known. After analysis, static code analyzers give a detailed list of the errors encountered as well as a description of the error and where they occurred. Tools issue warnings/errors about concurrency violations, implementation defects, boundary conditions, security weaknesses, logic errors, and other general defects [12] [1]. These tools tend to eliminate much of the process done during a code review, which is another form of static code analysis.

**Why to Use Static Code Analysis**

By integrating the use of a static code analysis tool into your development life cycle, you increase the likelihood of detecting safety and quality problems earlier, allowing you to remedy security breaches before they become a problem [12]. As shown in Figure 1, static code analysis quickly detects errors early in the coding stage, encouraging efficiency throughout the coding process [7].

Static analysis tools can prove to be a necessary part of software development. It is quite likely that various types of flaws exist all over your code. Static analysis tools can help to minimize their impact. These automated tools can be more efficient than code reviews or pair programming and consume far less time and resources, catching subtle errors easily missed by programmers. They may also be useful during maintenance of legacy code. For instance, if the developers had the habit of checking for NULL pointers in the past, it is possible that the current maintenance team may not be aware of such rules and will not do such checking. This type of problem may very well introduce some serious errors into previously functional code. In addition, static analysis tools will point out unclear code [2].

Through the use of static code analysis, you help to ensure that your software is of high quality and is thoroughly tested. These tools help detect less obvious errors, such as overflows resultant from the addition of two integers which will not be noticed by a compiler or by the programmer. It is quite possible that after adding two integers the result could be out of range, which will not throw an exception. Rather it will cause improper behavior of software which, depending on its use, could be fatal [1].

Many embedded systems nowadays work within networks, making them more susceptible to programming flaws and illegal inputs over the network. Traditional methods of code testing, such as code reviews, debugging, or test cases, fail to validate all likely paths of a program. On a similar note, the cost of creating thorough test cases is extremely high, and they still will not completely analyze a program. It is also usually the case that the costs of verifying, testing, and debugging far exceed that of the initial cost of developing the code base. In comparison, static code analysis offers the ability to verify all possible execution paths in addition to simply locating defects such as memory allocation errors, under/over flows, bound array access, and inconsistent code fragments.

One of the main selling points of static code analysis is that, unlike dynamic testing, it can be applied directly to incomplete or incorrect code that does not yet even compile, which can be done long before test cases are even developed. The effects of this can result in reduced time and costs, as well as increased revenue, and the reduction of business risk due to the production of reliable software [13].

**How Static Code Analysis is Used**

Static code analysis has a large variety of uses in the software industry.

**In Education**

***In Schools***

It can be used in school when a professor has to grade multiple student programs. The use of a static analysis tool can improve a professor’s grading accuracy by aiding in the discovery of issues that may be missed by human error. By finding such subtle errors, the professor is assisted in finding issues that certain students need to improve on [7].

***In Organizations***

Static analyzers can also play an important part in training new employees at the job site. The analyzer can be used by a senior employee to check the work done by the novice, helping them to find and explain errors in a much more timely fashion. In addition, such tools will also help new employees adapt to company standards and style, such as indents and variable naming conventions [7] [8] [12].

**Porting Software**

Porting software can be a large task, especially when it was not originally planned. It would be nearly impossible to know all of the issues that you may encounter when switching platforms, and it is not likely that it will be known how to locate dangerous code fragments. Static code analysis can easily point out these unsafe code fragments and will tell you where to modify code.

**Locating Suspicious Code**

Locating backdoor access points, areas of suspicious code that do not get executed under normal circumstances, is another concern when dealing with outsourced or third-party code. For this reason, a static code analyzer can prove to be a valuable asset by preventing attacks by hackers, who can use similar tools to help them locate such vulnerabilities. Open-source code can be run through such an analyzer to help decide which vendor to choose, based on which one has the least buggy or least dangerous code and is therefore the most safe to use.

**Code Refactoring**

Static code analysis may also be used to help determine if code refactoring may be needed. When projects become overly complex, they begin to fall apart. A static code analyzer can help by pointing out areas of code that need to be rewritten. These tools locate issues such as overly large functions, overuse of global variables, and complicated class hierarchies leading to unsafe functions. By drawing attention to these issues, they can be addressed before they become a much larger structural problem [7].

**Detection of Coding Errors**

Most static analyzers can be run after code compilation, alerting the programmers of possible errors [8]. If formal methods static analysis is incorporated as well, such analyzers can prove the absence of many bugs rather than just being able to locate existing bugs. Some issues that can be removed by these means include memory leaks and other pointers errors [14]. By finding and removing errors like these, static analysis tools can help build security and quality awareness leading to productivity gains, cleaner and more stable builds, and a higher quality product [12].

**Types of Static Analysis**

The definition of static code analysis is fairly broad, encompassing manual code reviews, style and error checking software, and more advanced formal methods-based analyzers.

**Code Review**

A code review is defined as reviewing source code in teams to reveal defects in other teammates’ code. The code should be relatively clear from the comments without assistance from the creator. Using a team to review code allows for better defect detection as it is usually easier to spot issues in someone else’s code than in your own [8]. Code reviews offer benefits that checking code dynamically does not. Code reviews can be used to locate a variety of issues, can offer a better understanding of the code after doing a review, and can be used to detect similar issues that static analysis tools locate, e.g. coding standards like MIRSA-C or JSF++ [6].

***Locating Vulnerabilities***

Code reviews can be used to aid in the location and detection of backdoors. By searching for code dealing with usernames and passwords, authentication logic can be checked to ensure its security. Similarly, backdoors can be prevented by ensuring that all testing functions, such as those that interfere or bypass authentication checks, have been removed before the product is released. Code reviews can also be used to identify malicious functions activated by certain parameters or those issuing unwanted system commands.

The act of malicious logging is another issue that can be detected by code reviews. Code reviews can reveal whether proper logging is enabled and where calls to logging functions, such as .NET functions, occur. By this process, improper logging can be uncovered by looking for the logging of sensitive data such as passwords, credit card numbers, or social security numbers.

The code review process can also check for proper methods of cryptography. It can be used to identify weak algorithms such as MD5, DES or SHA1 or custom-made algorithms as opposed to publicly trusted ones. Likewise, code reviews can ensure that applications are connecting to the database as a low-privileged user and that user credentials are not stored in plaintext.

Like other forms of static code analysis, code reviews trace data from source to destination, making it possible to identify which lines of code will cause vulnerability. The code review process should be done early in the life cycle, when it is most cost-effective and easiest to fix defects [11].

***Downsides***

Unlike the automated forms of static analysis, code reviews are far more time-expensive. Programmers need to be gathered at regular times to perform a code review. The team must be defined and people must be assigned to the positions of moderator, designer, coder and tester. In addition, a checklist needs to be created prior to review. There must also be scheduled breaks during code reviews to ensure that the reviewers do not grow tired after reviewing large segments of code, shortening their attention spans. It is also likely that a re-review will be required after the detected issues have been corrected. Code reviews also rely solely on the expertise of the reviewers [8] [6].

**Automated Tools**

The brunt of modern static analysis is done using automated tools. These tools have varying degrees of quality assurance and defect detection capabilities. Most commonly, static analysis tools only help in locating errors and coding defects or in meeting standards. However, there are the more advanced tools that use formal methods to prove the absence of certain bugs.

***Common Tools***

These tools automate much of the code review process, attempting to find errors in programs, but they do not guarantee their absence. They identify potential and actual defects in the code using heuristics and statistics that do not require execution of the code. Such tools can vary from variable initialization checks to data flow analysis. Although they may find some errors, they also have the issue of introducing false-positives and false-negatives. A false-positive occurs when reliable code is misidentified as erroneous, and a false-negative occurs when erroneous code goes unnoticed. Unfortunately, decreasing the probability of false-negatives increases the probability of false-positives [6].

***Formal Methods***

More advanced static code analysis tools, such as Polyspace, use what *formal methods* in their analysis. These types of tools are generally used in critical systems and medical software development. Faults may exist in the code but may only occur during a certain set of conditions, giving the appearance that the code is functioning normally. This may cause the application to fail or have fatal consequences. This technique of analysis uses mathematical concepts to find and attempt to prove the absence of critical errors, e.g. run-time errors. The main concept used in formal methods analysis is *abstract interpretation.*

Abstract interpretation uses mathematical theorems for defining rules that analyze complex dynamic systems. These rules help to prove the absence of detectable run-time errors, such as uninitialized variables, overflows/underflows, dividing by zero, and out-of-bounds pointers. The results of an abstract interpretation are given as: “proven”, “failed”, “unreachable” or “unproven”, for each operation analyzed. For instance, determining the sign of a computation is an example of abstract interpretation. The following example will show how abstract interpretation works in the tool called Polyspace.

Given code to analyze, the tool will search for places where a run-time error may occur. It then classifies them as *unproven* and then attempts to show they will *fail*, are *unreachable* or are *proven* not to fail (See Figure 2).

As shown in Figure 2, the results of the abstract interpretation are illustrated via color-coded underlined sections of code. In general, green indicates the absence of run-time errors such as overflows, dividing by zero, or illegally dereferenced pointers; red indicates areas with detected run-time errors; orange indicates areas that may fail under certain conditions; and gray show elements of conditional branches that cannot be executed (dead or unreachable code).

Formal methods analysis can be vital for improving the quality of embedded, high-integrity, and critical systems software, such as medical, avionic, and automotive software. They also help to reduce the possibility of false-negatives. Having a process that includes these methods of static code analysis can help organization attain a high quality process, helping developers with safety and quality practices. Formal methods analysis saves time and money by finding and eliminating defects when they cost the least to fix. Formal methods analysis gives knowledge as to whether code will or will not cause a fault and simplifies the debugging process by locating the source of a run-time error, eliminating wasted time searching for it [1] [6].

**Advantages**

Static code analysis offers many advantages over traditional methods of error detection. Its main advantage is that it reduces the price of fixing defects by detecting them earlier in the process, when they are cheaper to fix. It costs tens time more to fix a defect in the testing stage when compared to the coding stage. Adding static code analysis can result in reductions in development costs, higher revenue, and decreased business risk. Thus, less time is spent in maintenance and development, allowing for the product to be brought to the market sooner and stay longer. All of these help to increase company reputation and market position.

Static analysis tools offer full code coverage testing. They find defects in exception handing and logging systems and in rarely-used code, which are nearly impossible to locate by other means. Static analysis tools are not dependent on compilers or the environment in which the project is executed in. They help to locate undefined behaviors, which can occur when switching compiler versions or changing code optimizations. These tools can quickly and easily detect the consequences of the “copy and paste method”, saving time from trying to locate all copies manually [8] [13].

**Disadvantages**

Although static code analysis tools offer great advantages in development, they also have numerous drawbacks.

**False positive/negatives**

One major drawback to most simple tools, which do not use formal methods, is the probability of getting false positives and negatives. False negatives produce a false sense of security and allow bugs to be released. On the other hand, false positives can delay the release of a project and can result in unnecessary debugging and extra work. Also, simpler techniques cannot detect divide-by-zero errors which result from complex computations; they only flag them as a possibility, even if they never occur; since the tool does not know under what conditions such an error would occur [1].

**Integration issues**

Integrating tools into a company changes the way people work and must become part of the organization’s culture, integrating into developers’ daily work. Tools require careful integration into the build process, and although they offer great utilities to analyzing, they also require manual tuning. Static analyzers require investments in education and the time of the employees in order to learn the tools and get used to using them. One major issue in the use of static analyzers on legacy code is that vendors assume that the tools are to be used on a new project, but this is not the case much of the time. Companies may also be restricted by time or budget or are subject to client development methods and contractual obligations and thus will not be able to use such tools [12] [13] [10].

**Standardization**

Governments and industries have noticed the importance of safety, security, and reliability in software production and have done research to create standards for coding. Two of these standards, which deal mostly with embedded C/C++, are the following:

**CERT**

The Computer Emergency Readiness Team was founded by the U.S. government to research internet weaknesses, to identify frequent programming errors, and to create and educate developers in secure coding standards. The group found that most vulnerability originates from ordinary programming errors, so they made plans to lessen these issues. To address these errors, they created secure coding habits, which they named the CERT C/C++ Secure Code Standard.

**MISRA**

The Motor Industry Software Reliability Association has developed guidelines for use in critical systems. The association’s guidelines deal with the worldwide automotive industry, including aerospace. In the past, they’ve created guidelines for C, but recently they have developed guidelines for C++ as well.

Many static analysis tools have been upgraded to meet both of these standards [13].

**Tips for Integration**

Using a static code analyzer on legacy code for the first time is likely to detect thousands of issues. These issues have probably existed for quite some time and have not adversely affected the software. You should have a plan in place to deal with issues that will be found. It may be a good idea not to tell developers of these issues so as to allow them to focus on the prevention of new issues. These issues can be deferred until a later time when they can be reviewed and remedied in a future release. Have developers do build analyses often; if they have incorporated dealing with these issues into their regular routines, then no problems should be found at this point [12].

It is important to focus on preventing future errors from occurring and on cleaning up code that is going to ship to customers [2].

**Subject Matter Experts**

It is advisable to create several subject matter experts (SMEs). It will be the duty of these experts to learn how to use and service the tools as well as how to educate the developers on how to use them. They will also be in charge of answering questions, such as assisting with identifying *false-positives*. SMEs should be assigned to every product or divided up in the case of larger products. It is necessary that they understand their responsibilities and the time that the job requires. They should be considered experts on their assigned tools and will be in charge of integrating the use of the tools into the daily work of the developers, getting them used to analyzing code prior to check-ins and fixing any found issues [12].

**Deciding on a Tool**

There are numerous free and commercial static code analysis tools available. When deciding on a tool, it is important to note that they cannot be compared by the number of defects they can find. The field of static code analysis is actively advancing. New rules and standards are created while others become obsolete. It is more viable to focus on the number of actual errors detected by various tools on a set of projects [8].

Some tools focus on a single language while others focus on multiple. When looking at tools, you should keep in mind what is best for your organization; this may include quality and security checking abilities, standards used, cost, licenses, or how easily you can integrate it into your process [10]. ACI Worldwide suggests asking five questions when deciding: “Do you need a static or dynamic analysis tool? What languages and platforms does it support? How flexible is the reporting component? How easy is it to add or update rules? Does it integrate with your IDE?” [12].

**Comparing Tools**

When comparing tools, there a number of things you should consider. The number of diagnostic rules does not mean that a tool is better than another. If you only use certain operating systems and compilers, then rules pertaining to others are irrelevant to you and only bog down the process of integration and complicate setup. The number of rules is in no way related to the number of errors a tool can detect. For instance, an analyzer designed for Windows applications will find more errors in a Visual Studio project than a cross-platform analyzer will find with 1000 rules. It is also not wise to decide solely by comparing rules in system specific tools. Each tool may have divided up rules in different ways, but still end up detecting the same amount of errors; they may also be of different quality levels. The speed of processing code also does not give a good comparison, since most builds are done nightly. As long as the analysis is done in the morning, it is irrelevant how long it takes.

It may prove more valuable to compare tools based on usability. The following example shows some usability differences between Visual Studio’s built-in analyzer and the PVS-Studio tool. PVS-Studio, unlike Visual Studio, gives the ability to filter duplicate warnings, save errors for review later, allow users to specify which files are to be analyzed, easily hide and reveal errors without re-analyzing the code, and filter errors based on containing text. While both tools can detect the same amount of errors, it is clear that usability of tools factors into the decision. Comparing two analyzers is a complicated task and should be all about what tool is best for its users and the project [9].

**Tool Examples**

***IntelliJ IDEA***

IDEA is designed to improve code structure, conform to guidelines and standards, detect performance issues, and errors. It can be used to locate inconsistencies, probable bugs, redundancies, specification violations, and others. While using the tool, errors are highlighted in the code as it is being written making it easier to use without interrupting the coding process. IntelliJ IDEA is a standalone IDE with a built-in static code analyzer [5].

***Visual Studio’s Integrated Code Analysis***

Visual Studio’s static code analyzer has over 200 rules broken into different groups, which are further broken down into rule sets aimed to target specific coding issues. It can be used as part of an automated build process, can be run manually, or can be part of a check-in policy. By default, detected errors are displayed in the Errors Window as warnings, although this setting can be changed. Each error has detailed information, the code file it was found in, and which line it occurred on. Violations have the option of being fixed or suppressed when they are not applicable [4].

**Conclusion**

Static analysis tools can prove to be a valuable tool in error detection in the process of software development. They have numerous advantages and functions when incorporated with organizations’ processes. Although they may be rather troublesome to integrate at first, the benefits they provide in terms of cost and time saved is rather significant. Most tools nowadays conform to current coding standards and can be used to enforce such company policies. In the end, however, it all comes down to what tool and type of static code analysis has the best fit with a company and how they want to use it.

**References**

[1] Abraham, J. (2012, June 6). *Using formal methods for sophisticated static code analysis*. Retrieved June 25, 2012, from EE Times: http://eetimes.com/design/ embedded/4374801/Using-formal-methods-for-sophisticated-static-code-analysis

[2] Carmack, J. (2011, December 27). *In-Depth: Static Code Analysis*. Retrieved June 25, 2012, from Gamasutra: http://www.gamasutra.com/view/news/39328/InDepth \_Static\_Code\_ Analysis.php

[3] Gousset, M. (2010, April 27). *Static Code Analysis Configuration*. Retrieved June 27, 2012, from Visual Studio Magazine: http://visualstudiomagazine.com/articles/ 2010/04/27/static-code-analysis-configuration.aspx

[4] Gousset, M. (2010, March 25). *Static Code Analysis in VS2010*. Retrieved June 25, 2012, from Visual Studio Magazine: http://visualstudiomagazine.com/articles/ 2010/03/25/working-with-static-code-analysis.aspx

[5] JetBrains, Inc. (n.d.). *Static Code Analysis*. Retrieved June 25, 2012, from JetBrains: http://www.jetbrains.com/idea/documentation/static\_code\_analysis.html

[6] Jones, P., Jetley, R., & Abraham, J. (2010, February 9). *A Formal Methods-based verification approach to medical device software analysis*. Retrieved June 27, 2012, from EE Times: http://eetimes.com/design/embedded/4008888/A-Formal-Methods-based-verification-approach-to-medical-device-software-analysis

[7] Karpov, A. (2010, December 27). *Cases When a Static Code Analyzer may Help You*. Retrieved June 25, 2012, from The Code Project: http://www.codeproject.com/ Articles/ 140078/Cases-When-a-Static-Code-Analyzer-may-Help-You

[8] Karpov, A. (2012, March 12). *Static code analysis*. Retrieved June 25, 2012, from CPlusPlus.com: http://www.cplusplus.com/articles/9E18T05o/

[9] Karpov, A., & Ryzhkov, E. (2011, March 31). *Difficulties of comparing code analyzers, or don't forget about usability*. Retrieved June 28, 20120, from viva65: http:// www.viva64.com/ en/a/0071/

[10] Pitchford, M. (2011, March 1). *Think static analysis cures all ills? Think again.* Retrieved June 25, 2012, from EE Times: http://www.eetimes.com/design/ embedded/4213633 /Think-static-analysis-cures-all-ills--Think-again-

[11] Shetti, V. (2010, August). *Why Static Analysis?* Retrieved June 25, 2012, from Palizine: http://palizine.plynt.com/issues/2010Aug/why-static-analysis/

[12] Sidner, S. (2010, April 24). *When Quality, Security Count*. Retrieved June 25, 2012, from Dr. Dobb's: http://www.drdobbs.com/ tools/224600102

[13] Vink, G. (2010). *Static Code Analysis (SCA) Standardization Efforts & Integration in the Software Development Flow*. Retrieved June 25, 2012, from Tasking: http://www.tasking .com/resources/Static-Code-Analysis-WhitePaper.pdf

[14] Yocum, C. (2011, May 14). *An introduction to static code analysis: What, why and how*. Retrieved June 25, 2012, from The Register: http://www.theregister.co.uk /2011/05/14/ static\_code\_analysis\_101/

[15] Zakhareyev, E. (2008, February). *Microsoft Static Code Analysis Tools Survey*. Retrieved June 25, 2012, from Attrice: http://www.attrice.info/articles/Static CodeAnalysisTools Survey.htm

**Appendix**

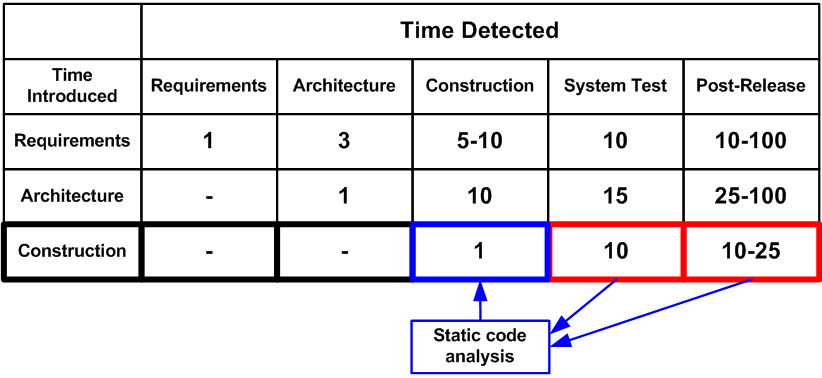


Figure 1: Average cost of fixing a defect depending on when they are detected [8]

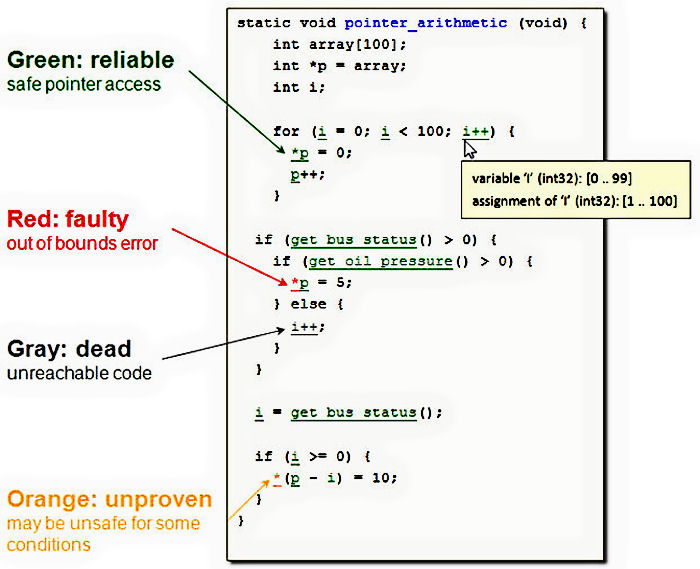


Figure 2: The results of an abstract interpretation in Polyspace [1]