



PACIFIC NW  
28TH ANNUAL  
SOFTWARE  
QUALITY  
CONFERENCE

OCTOBER 18TH – 19TH, 2010



ACHIEVING  
QUALITY  
IN A COMPLEX  
ENVIRONMENT

*Conference Paper Excerpt  
from the*  
CONFERENCE  
PROCEEDINGS

---

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

# Using Static Code Analysis to Find Bugs Before They Become Failures

**Brian Walker**

Tektronix, Inc.  
Brian.R.Walker@tektronix.com

## **Abstract**

Automated static code analysis tools have evolved considerably from the original lint tool. They are now more comprehensive, provide more relevant results and produce fewer false positive reports. The Video Product Line organization at Tektronix has integrated static source code analysis into its nightly build process. The analysis tool has identified several errors that produced faulty behavior or crashes which could have been avoided. It has also identified subtle errors that were overlooked by human observers or simply not covered by functional testing. In concert with automated functional testing, static source code analysis has enabled the Video Product Line to achieve better quality and devote more time to feature development.

## **Biography**

*Brian Walker is a Senior Software Engineer and software architect in the Video Product Line at Tektronix. He has over 15 years of experience in embedded software development using both VxWorks and Linux. Brian started his career at Tektronix as the resident expert for DDTS, ClearCase, VxWorks and other software tools. Brian received a B.S. degree in Computer Science from the University of Colorado at Boulder.*

## 1. Introduction

In the beginning, there was lint. Lint was developed as a tool to improve the portability of source code written in the C programming language. Lint primarily checked the syntax of a program for known weaknesses of the C programming language and provided warnings about undefined behavior, uninitialized variables and non-portable code. Later variants also validate the number and types of arguments passed to functions. These checks were useful and helped prevent a number of common defects. It is no wonder then, that many of the checks that lint performed have been integrated into compilers.

Lint, itself, is a program for static source code analysis. In short, static analysis is the process of analyzing a program without actual execution. Its converse, dynamic analysis or dynamic testing, is the process of testing a program through its execution, such as through automated regression testing, expert user testing and acceptance testing. Static code analysis also includes code reviews, but this paper will specifically focus on the use of automated static code analysis tools to improve the quality of software.

Two years ago, my department purchased Klocwork Insight, a commercial static analysis tool. We integrated the tool into our nightly builds and began performing automated static analysis in addition to our automated regression tests. The tool has provided us with a significant list of issues within our source code which has guided our efforts to improve the quality of the software.

## 2. Beyond Lint

The popular definition of static code analysis now extends beyond the basic lint functionality and usually includes data and control flow analysis. Static analysis can examine not just the structure of a function but also its interactions with functions that it calls and functions that call it. Rather than examine a single file, it can examine the file in the context of how it is used and how it uses data from functions it calls. A good static analysis tool can follow data through potential paths of execution and identify where it is allocated, initialized, used and eventually discarded. The analysis can be performed line by line so that each line of code is analyzed and dead code is identified. That analysis, by necessity, must be able to follow the data even as it passed by reference to other functions. When it finds a problem, the tool, can then identify the path required to produce that error.

Unlike dynamic testing, static analysis can provide nearly full coverage because it does not actually have to exercise the program to get a result. By analyzing the possible set of values of variables at each line a function, it can determine where those values would produce an error such as a null pointer dereference or an array index that is larger than the size of an array or even a negative value. The static analysis provides a list of possible errors and locations where certain possible values will produce an error.

Since the static analysis tool typically relies on a pattern library to detect errors, like any expert system, it is only as smart as its teachers. Static Analysis is not perfect and will miss some bugs. And if the analysis does not go deep enough into the code, it will miss instances where the likely errors are prevented or avoided. So, the designer of a static analysis tool must balance the aggressiveness of the process for finding potential errors with the restraint to avoid false positives. So, if the tool does not produce at least a few false positives, it's probably not trying hard enough.

One of the problems with basic lint was that it would produce a large amount of warnings. The poor user, who was inundated by these warnings, might be so overwhelmed that he would simply stop using the tool or sheer number of warnings might well hide the truly significant errors that needed to be addressed. When reviewing a popular lint program, I once noted that one of the benefits of the program was that it was highly configurable and that any of its messages could be disabled in a very flexible manner. The problem with such a tool was that it required a lot of manual intervention to tailor the results. Many of the warnings were useful but having too many warnings reduces the utility of the tool.

### 3. Elements of a Static Analysis Tool

There are four useful features of a static analysis tool. The first, of course, is an extensive library of patterns and rules for detecting issues. It should also have a database for tracking reported issues and an interface for reporting those issues to the user. Finally, it needs a compilation process to perform the actual analysis.

The pattern library includes the analysis engine plus all of the rules and patterns used to detect errors within the software. Since not all issues are created equal, it should identify problems by severity so that users can prioritize issues, especially when the analysis is run for the first time. The library should contain patterns for a wide range of issues including potentially catastrophic errors to simple warnings. Preferably, the library also includes the ability to enable or disable patterns in the library so that users may customize working of the tool.

A database for tracking known issues should be considered an essential part of the tool. No static analysis tool can be perfect. At best, a static analysis tool provides a good approximation of the errors a program might encounter. When in doubt, the tool should error on the side of caution in order to alert users to possible problems. That naturally results in a number of false positives which, unless instructed to ignore, the tool will continue to report. The tracking database can also identify the problem more succinctly than by just the line number and continue to track the issues as the source files changes.

A static analysis tool should also have a good reporting interface so that the user can quickly determine the nature of the problem, and more importantly, how to fix it. Having the reporting interface integrated into a source code browser with a cross reference capability allows a user to investigate the problem quickly and determine the accuracy of the report. In some cases, that reporting capability might be integrated into the IDE environment.

Finally, the tool must have a function to analyze the source code and generate reports. This often takes the form of a compilation and linking process that analyzes the structures of files and combines the information with other files to produce a result. The compilation process may require as much time as the compilation and build of the executable. The tool must run the analysis with the same options and compiler flags as the normal software build so that the analysis and the build see the same lines and values within the source code.

Some have proposed that static analysis could be integrated into the compiler. There is ample reason to support tight integration of the static analysis within a compiler. For one, it absolutely ensures that the compilation and the analysis are performed on the same code. The analysis can also be more efficient by leveraging the analysis performed by the compiler's optimizer. The analysis can be seamlessly integrated into the normal build process and the results can be displayed directly in the IDE. Both Apple Computer and Green Hills Software offer such a solution.

But, as an embedded systems developer, I first need to find a compiler that supports the target processor I am using. A tight integration of compiler and analyzer would limit my choices of compiler. Therefore, there is still a need for stand-alone static analyzers.

### 4. Using the Klocwork Static Analysis Tool

The process supported by Klocwork automates the generation of a build manifest by running the normal build process within a manifest generation tool that captures the build commands from the normal build script. The advantage of this approach is that it provides a simple and generally universal methods to capture the list of files and commands used to build the software. However, it does not support incremental builds so using this approach with the Klocwork tool requires either performing a full build every time or managing the build manifest separately and manually rebuilding the manifest as needed when the build configuration changes.

Since we use ClearCase, I chose the third option which is to harvest the build manifest from ClearCase configuration records. We use Clearmake to perform all of our builds which performs an audited build within ClearCase and records a complete transcript of the build including all files and commands used to build each object and assemble them into programs and packages. The format of the build manifest file is clearly described in the Klocwork documentation so I developed a simple script to extract the commands from the ClearCase configuration record and generate a build manifest for Klocwork. The use of ClearCase configuration records allows us to fully automate the static analysis process because the configuration records are maintained by ClearCase for all builds and are always complete even after performing an incremental build.

The next step is to generate a record of file ownership. Establishing ownership is not strictly required for the static analysis but it does help document responsible parties for issues that are discovered during static analysis. One way to organize ownership is by module or component where people have specific responsibility for parts of the software. In our environment, the owner of the file is simply the last person to check it in. Once again, product documentation describes the format of the file and a simple script extracts the information out of ClearCase.

Using the build manifest, Klocwork performs a build. It identifies the compiler commands through the use of a filter file which provides the names of the compilers and linker for several variants of compiler tool chains. In our case, we needed to add the names of the GNU compilers and linker because the names of the cross compilers we use are different for each target processor and often subject to the whims of the tool chain provider.

The analysis requires at least as much time as the actual build and runs in several phases. The results are collected in a table which, when complete, is uploaded to the database. The results of the build are then available through the reporting interface. This interface provides access to all of the issues recorded in the database. Klocwork provides the reporting interface as a web service allowing any user to access the results from anywhere, including my home computer.

## 5. Managing Issue in Klocwork

Klocwork records the state of an issue using two fields. The “State” field is the state determined automatically by Klocwork which may identify issues as new, existing, recurred, fixed or not in scope. As suggested by the states, Klocwork maintains a history of current issues and has the ability to detect when new issues are discovered and when they are fixed. The “not in scope” state means that the file containing an issue was not in the previous build. That may occur when a file is removed from a build or if a file was inadvertently not included.

The “Status” field is a state assigned by the users and reflects the priority of an issue. Every new issue starts in the Analyze state. In the Klocwork process flow, issues in the Analyze state are categorized as “uncited” with the expectation that a reviewer will cite the problem by setting the Status field to some other state. Since Klocwork is run as part of our nightly or weekly builds, I usually spend a few minutes each morning reviewing the results from the previous night. If Klocwork discovers new issues, I review the issue reports and change its status to Fix, Fix in Next Release, Fix in Later Release or Not a Problem. Issues that are in the analyze or fix state are presented as open issues and are selected by the default filters. Any user can also add a note to the issue record and Klocwork maintains a history of both the status and notes.

One of the strengths of Klocwork is the depth of the information it provides in issue reports. An issue report contains a description of the problem, usually with specific description such as “Buffer overflow, array index of ‘foo’ may be out of bounds. Array ‘foo’ of size 22 may use index values of -2 .. -1. Also, there are similar errors on lines 491, 494.” The report also provides a traceback showing all locations where a value was set or used and conditions required to get to the location where the error may occur. Due to the nature of the analysis, that backtrace will often include source lines in other functions when the value in question is returned by a function or passed to a function. The lines referenced by the

traceback are highlighted in the source listing where they can be examined in context. Clicking on a traceback item positions the source listing at the line and highlights the line in the source listing.

The source listing also provides a cross reference interface that I can use to explore the source code. With it, I can quickly find declarations for variables, macros, constants or functions or find the implementation for a function. I can also find other places in the source code that use a class or function or are used by the class or function. The ability to quickly dive into a problem is a welcome feature of the Klocwork user interface and helps me diagnose issues or fix issues more quickly. After all, using static analysis is supposed to be saving me time and the quicker I can find the cause of a problem, the quicker I can resolve it.

Klocwork also provides direct access to the cross reference interface. That is useful for finding all of the issues found in particular files or directories. That way, if I am editing a particular file, I can use the cross reference to find all of the issues reported in a file and fix them all at once. The cross reference also provides information on the structure of the software by providing references to related objects, functions and constants. It also provides a reverse engineering capability that can help a developer understand the code.

Unfortunately, Klocwork does not integrate with our defect tracking system, or any defect tracking system. Although I can reference any issue in the database with a URL, it would be more useful if I could simply click a button and conveniently submit the issue into our defect tracking system and even more convenient if it could automatically resolve the defect in our defect tracking system. Usually, I just cut and paste the URL and send it in an email message.

## 6. Types of Issues Found by Static Analysis

Klocwork, like many static analysis tools, can detect several well known and preventable errors. Some of the types of problems that it detects include null pointer dereferences, array buffer overruns, memory leaks, use of freed memory, file resource leaks, concurrency violations, uninitialized data and security vulnerabilities.

Klocwork uses a convenient short hand notation to identify each problem type. For example, ABR indicates a general array buffer overrun while NNTS identifies a non-null-terminated string. The notation tends to be concise and memorable so I can often determine the type of problem in a glance. If in doubt, simply clicking on any of the notation in the web interface provides the description of the problem in the online documentation. That documentation tends to be well written and provides a description of the problem, specifies vulnerabilities, identifies risks associated with the problem and often includes instructions for mitigation and prevention of the problem.

Klocwork also provides several flavors of issues. One common distinction are the classifications of “must” and “might” which indicates whether a potential error depends on one or more factors. A “must” issues indicates that the error will occur as a results of a single failure. A “might” issues indicates that a failure might occur if more than one condition is true. For example, an NPD.FUNC.MUST issues describes a situation in which a pointer, returned by a function that could return NULL, is always dereferenced by the caller. An NPD.FUNC.MIGHT issue describes a situation in which the caller might dereference that pointer if certain other conditions are also true.

Klocwork has been very effective in discovering null pointer dereference issues within our source code. Many of these issues are the results of source code that does not check the values of pointer parameters passed into a function or the pointers returned by a function. Klocwork can identify several varieties of null pointer dereferences depending on where the pointer dereference occurs and whether the program actually checked the pointer. For example, NPD.CHECK.MIGHT describes an issue in which a pointer was previously checked for NULL but the program dereferences the pointer anyways. This problem can be quite insidious because it can easily be missed in a code review. Reviewers who are not paying attention may see the check for NULL but miss the nuances of the implementation. In the following sample, the pointer pdu is checked for NULL but only if reqid != state->reqid.

```

int handle_request( int op, int reqid, PduType *pdu, SyncState *state)
{
    if (reqid != state->reqid && pdu && pdu->command != MSG_REPORT) {
        return 0;
    }

    if (op == RECEIVED_MESSAGE) {
        if (pdu->command == MSG_REPORT) {

```

There are also several flavors of array buffer overflows and Klocwork has discovered quite a few in our source code. String handling is a common theme in the issues reported by Klocwork such as using `strcpy()` to copy strings of different or unknown sizes. One interesting scenario that Klocwork has reported involved the use of `sprintf()` to construct a string. Klocwork reported an issue because the programmer had sized the buffer to fit the expected result, but not large enough to handle unexpectedly large numbers or long strings. Another common error involves iterating through an array using the wrong constant as an upper bounds for the array index.

Most memory leaks are of the `MLK.MIGHT` variety and are often caused by improper error handling. The typical scenario involves the handling of errors conditions where the code returns from a function without freeing memory that was previously allocated by the function. These sections of code are typically not covered by normal functional tests because the errors they handle are not encountered under normal conditions. Unit testing might cover this sort of failure if they were written and if the programmer was diligent about testing all paths through a function. Memory leaks can also be difficult to detect because memory is often lost a little at a time. That may not matter in an application running on a desktop with massive amounts of virtual memory at its disposal. But, if your goal is 24/7 operation in an embedded device, those memory leaks tend to add up to produce subtle failures and memory fragmentation.

Another place that Klocwork looks for memory leaks are in class object initialization. If a C++ class allocates memory in the constructor, Klocwork will verify that the memory is freed by the destructor. Likewise, if memory is deallocated in the destructor, Klocwork will warn of the potential for double freeing of memory if the class does not define both an assignment operator and a copy constructor.

Klocwork includes several checkers for use of memory after it has been returned to the system heap. Prior to running Klocwork, I did not expect to see this particular issue in our software. Surprisingly, Klocwork has reported this issue in third-party and open source software that we use in our product.

File resource leaks, like memory leaks, are often caused by improper error handling but the pool of file handles is more limited. However, resource leaks are given a lower severity by Klocwork. While they are reported as more of a warning, than as an error, Klocwork has found a number of locations in which file handles could be lost.

Since our software is multithreaded, the ability of Klocwork to detect concurrency violations has been useful and has uncovered problems where semaphores could be acquired indefinitely. Klocwork provides warnings when semaphores are taken within a function but not unlocked when the function returns and in situations where a thread goes to sleep while a semaphore is locked.

I expect that any static analysis tool should be able to detect use of uninitialized data. Often, it appears, people simply forget that automatic variables within a function are not initialized to a specific value.

Security vulnerabilities are becoming a much greater concern as more devices become connected. However, secure programming practices are less obvious and easily overlooked. A vulnerable program will still pass it's functional tests and the vulnerabilities will not be exposed until someone specifically tests for them or tries to exploit them. The number of reported vulnerabilities in software continues to grow and the cost to fix them in the field may not only include the time and expense to fix the defect but also carries the potential for exposure of sensitive data and erosion of customer trust. Most vulnerabilities can be traced to a relatively small number of common programming errors (Mitre/SANS 2010). This an area

where prevention is far more effective than remediation and regularly scanning for these common errors can reduce or prevent most security vulnerabilities.

Many of the issues in the Klocwork pattern library can be mapped to the Common Weakness Enumeration dictionary sponsored by the National Cyber Security Division of the U.S. Department of Homeland Security (Klocwork 2010). Klocwork analyzes code for several specific types of security vulnerabilities including unvalidated user input, SQL injection, path injection, cross-site scripting, information leakage, weak encryption and other vulnerable coding practices. Unvalidated user input can cause a whole host of issues including buffer overruns, injection of executable code, spoofing and compromised programs.

Klocwork recently introduced support for the MISRA C and C++ coding standards. The MISRA C and C++ standards were created by the Motor Industry Software Reliability Association to facilitate safety and portability of software written for automotive control systems. These standards have also been adopted by developers in the aerospace, telecommunications, medical devices, defense, railway and other industries (MISRA 2009). The standards define a coding standard composed of over a hundred rules to make software more reliable.

## 7. Analyzing the Results

After running the Klocwork tool on our code base of about 500,000 lines of C and C++ code, the tool produced a list of about 1500 issues. After reviewing those issues, I eventually rejected about 175. The remaining issues were significant problems and, in some circumstances, blatantly incorrect. In my judgement, the total number of issues reported and the number of false positives were quite reasonable.

Unfortunately, the problem with reporting 1500 issues in an established code base is that someone must go through those reports, categorize them and actually fix the problems. Introducing a static analysis tool to an established code base will most likely create a sizable bug debt. However, once the bug debt has been paid, the true value of static code analysis will be that the problems found can be quickly identified and corrected even before they are tested or delivered to customers. Static analysis is most useful when it is employed on new new software as a preventative measure.

There have been instances where I have literally spent hours trying to reproduce and capture an instance of a crash bug with a debugger and then found that Klocwork had found that same bug, identified its location and the conditions necessary to produce the crash. In one case, the crash was due to a race condition that was only triggered under certain specific conditions and in a particular order. I spent most of that time trying to reproduce the failure hoping to catch it in gdb. Had I checked the Klocwork database and fixed the bug as soon as it was reported, I could have avoided the grief.

Other bugs express themselves in less subtle ways. Most of the time, when I have encountered bugs that are the result of a stray pointer, the cause tends to be a simple array buffer overrun. Sometimes, the array just happens to be in memory adjacent to a data structure that is used by a different thread so using a debugger to step through the code does not always find the culprit. In some cases, the stray pointer writes over a virtual table causing the program to crash mysteriously. Finding such a bug requires first finding the location that gets overwritten and then setting a watchpoint to trap the write. Finding it with static analysis can literally saves days of aggravation.

A substantial number of the issues found by static analysis were in error handling and were missed by functional testing precisely because they do not normally occur in the normal execution of the application. Some would have been caught by unit testing had the programmer written a test to stimulate that path. Unfortunately, the error eventually does occur and instead of gracefully recording the error and continuing, the application would unceremoniously crash. Hopefully, that moment of failure occurs before the software is delivered to customers.

Another advantage of static analysis tools is that they also promote good programming practices. Many of the issues discovered by static analysis go back to basic principles like check those pointers for NULL

before dereferencing them, check your array index before using it to access an element in an array or close that file handle before returning an error from the function.

## 8. Looking Forward

Currently, my team has only integrated static analysis into the nightly builds. The results are available on a daily basis. An even better integration would be to integrate the static analysis into our development environment and provide direct and immediate feedback to all of our developers. Klocwork and other vendors of static analysis tools offer integrations for both Eclipse and Microsoft Visual Studio.

Conceptually, we would continue to run nightly builds to perform the bulk of the analysis. The client tool could then link to the shared database, run the analysis on the files currently checked out in the user's sandbox and cross reference the information to provide results for the software as it is currently configured in the user's environment. That capability would allow the user to receive immediate results before committing changes to the version control system. Eventually, the static analysis could be used as a gate to checkin of the files into the version control system and thereby prevent coding weaknesses from leaving the developers desktop.

## 9. Fitting it all Together

Static analysis is an excellent tool for finding problems in source code early in development. The ability to analyze code without execution enables software to be analyzed before it can be loaded and run on an actual hardware. It can provide more complete coverage of the source code than dynamic testing for a finite set of known problems because it is able to follow all paths within each function without reliance on external stimuli. Testing and analysis are adept at finding different kinds of problem. For one, static analysis cannot determine if the operation of a software program conforms to the product requirements. And, testing cannot accurately or repeatedly determine the structural soundness of the software.

Likewise, static analysis can complement the practice of formal and informal code reviews. While human reviewers can make mistakes and overlook problems, static analysis can consistently detect a known set of common weaknesses. It's kind of like having an older brother who is always willing to tell you about your faults. But static analysis cannot replace expert eyes of peers who not only have the imagination to look beyond the code but also have an understanding of the product requirements and the experience to offer better solutions.

Sometimes, there really is no other option. Most software development uses open source and third party software. Short of assigning an engineer to review all of the code or write tests to thoroughly test the code, how does one assure that the software is free from defects? Some open source communities invest in test suites but many do not. Most open source software is provided "as is and without warrantee" according to the terms of the GPL or other open source license. In my team's product software, more than half of the remaining issues in the Klocwork database were detected in an open source library.

Automated static analysis is not a silver bullet but it can provide immediate feedback to help write better software. It can increase the tribal knowledge of an organization by introducing a knowledge base of known software weaknesses and can provide information on how best to avoid them. If used frequently, it can allow an organization to find and fix those problems before testing and before they become bugs that annoy customers.

## References

MITRE/SANS 2010, "CWE/SANS Top 25 Most Dangerous Software Errors" MITRE Corporation, <http://cwe.mitre.org/top25/index.html> (last modified June 29, 2010)

Klocwork 2010, "Detected C/C++ Issues" Klocwork Inc., [http://www.klocwork.com/products/documentation/Insight-9.1/Detected\\_C/C%2B%2B\\_Issues](http://www.klocwork.com/products/documentation/Insight-9.1/Detected_C/C%2B%2B_Issues) (last modified May 18, 2010)

Klocwork 2010, "Detected Java Issues" Klocwork Inc., [http://www.klocwork.com/products/documentation/Insight-9.1/Detected\\_Java\\_Issues](http://www.klocwork.com/products/documentation/Insight-9.1/Detected_Java_Issues) (last modified April 16, 2010)

MISRA 2009, "MISRA C" MISRA Consortium, <http://www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx> (accessed August 15, 2010)

Coverity, "Resource Library, Case Studies" Coverity Inc., <http://www.coverity.com/html/research-library.html#CaseStudies>

Various Authors, "Static Code Analysis" Wikipedia, [http://en.wikipedia.org/wiki/Static\\_code\\_analysis](http://en.wikipedia.org/wiki/Static_code_analysis)