

GIVE
YOUR
DEFECTS
SOME

USING
AUTOMATED

STATIC ANALYSIS
TO
DEBUG

BY GREG POPE, KIM FERRARI, AND BILL OLIVER



ALYZERS

YOUR
CODE

MANY of this magazine's readers are familiar with the *STAREAST* and *STARWEST* conferences. Here is a trivia question for you—what does STAR stand for? Give up? It stands for Software Testing Analysis & Review. It is an acronym for the types of activities that we in the software quality field perform. However, in many organizations, the analysis activity is often overlooked, and that's too bad because analysis is a powerful tool in the quality arsenal.

Static code analysis is computer software analysis that is performed without actually executing that software. (Analysis performed on executing software is known as dynamic analysis.) In most cases, static analysis is performed on the source code. In recent years, the importance of computer security has created an expanded demand for automated tools that can analyze source code for security vulnerabilities and coding defects that could be exploited. Many security vulnerabilities are caused by questionable coding practices—for instance, using an input variable as a loop index without first checking that its value is within a valid range. Contemporary static analysis tools are able to analyze source code with a much lower false-positive rate (claiming code is defective when it is not) than previous lint-style detector tools. Because they examine only small portions of the source code at a time, lint-detector tools typically have false-positive rates of 50 percent or higher. The leading contemporary automated static analyzer (ASA) tools claim—and our experience to date has shown—false positive rates under 20 percent. These tools achieve this by parsing the source code in a way similar to compilers, creating a syntax tree and database of the entire program's code, which is then analyzed against rules or models. The ASA tools then create a report of suspected defects in the code.

Requirements	8.1%
Features and Functionality	16.2%
Structural Bugs	25.2%
Data	22.4%
Implementation and Coding	9.9%
Integration	9.0%
System Software Architecture	1.7%
Test Definition and Execution	2.8%
Other	4.7%
Sample size	6,877,000 statements (comments included)
Total defects	16,209
Bugs per 1000 statements	2.36

Table 1

Many of the defect types found by ASA tools would be difficult to find using peer group inspection techniques (the R in STAR). This is because a defect may be a combination of source-code statements that are physically far from each other in the source code—for instance, the allocation of memory and then, pages later, a return statement without releasing that memory. A human is limited in the amount of source code-detailed information that can be remembered from one page to the next. An ASA is not limited by this restriction, and besides, most of us do not relish the prospect of examining other people’s code for hours at a time. ASA tools also can find problems that elude traditional system-level testing—for example, an array-bounds overflow, where a string of twenty characters is written into a buffer of size ten. The ten memory locations that are overwritten may not manifest a failure in the program until a later execution time or may not manifest in a repeatable way.

Since ASA tools are totally unaware of user requirements, the tools cannot replace the benefits of peer reviews or good functional testing. Also, ASA tools will not replace the need to use dynamic

analyzers. Dynamic analyzers can find problems that occur only during interactions between the executing application code, system resources, and interfaces such as race conditions.

While ASA tools are not a “silver bullet,” these tools have the capability to detect up to 50 percent of defect types and security vulnerabilities before system testing is conducted, which reduces the amount of time needed for system testing and reduces the risk of defects’ escaping to the field and being discovered by customers. Table 1, from Boris Beizer’s *Software Testing Techniques* [1], lists typical defect types and their percentages:

ASA tools are effective on structural bugs (which are approximately 25 percent of all bugs); 66 percent of data bugs (about 14.5 percent of all bugs); 66 percent of implementation and coding bugs (about 7 percent of all bugs); and 50 percent of integration bugs (about 4 percent of all bugs), for a total of approximately 50 percent of the types of bugs found in systems. Note that ASA tools cannot find defects related to requirements, features and functionality, and testing defects.

Because static analyzers use parsers, they are limited to specific source code

languages that they can analyze. The most common languages handled by ASA tools are C, C++, and Java. Some ASA tools can handle multiple languages because they have multiple parsers. The ASA tools with the fewest false-positive rates also compile and link the source code. Reports generated by the ASA tools assume that ASA users have a basic familiarity with the source language being analyzed. ASA tools can be used for newly developed code as well as for legacy code. They can be used as a risk-mitigation strategy for both open source code and vendor-supplied source code by incorporating ASA tools into an acceptance test procedure or release criteria. ASA tools also can be used on source code generated by automated code generators, such as code that is generated from graphical models.

As use of ASA tools becomes widespread in software quality assurance organizations, the requirement for test engineers to become familiar with the source languages used will also increase. In the early days of software development, most of the developers (who also did the testing) were of necessity familiar with the languages used. In the last three decades, specialization has evolved into developers who write code and test engineers who test the integrated code. Black box test engineers traditionally specialized in domain-specific knowledge or business requirements and were generally discouraged from understanding what the code did. The belief was that an understanding of the code (white box) would bias the thinking of the tester. Instead, the tester should have the same perspective as the customer.

ASA tools present an ideal reason for testers to upgrade their skills and learn programming languages—not so they will understand the interworking and logic of the code under test, but so they can understand the ASA tool’s defect report and explain its findings to developers in a common vocabulary. Testers can use their language skills to prefilter the ASA tool’s report to remove any false positives or defects that are superfluous in the context in which the code is used.

A common example of this is a call to an error-handling routine that aborts the execution of the code. Since the error

routine does not return to the calling code, an ASA-reported error of “not checking the error flag” after the error routine returns would be superfluous. Many ASA tools have the ability to be trained to ignore these types of implementation-specific errors. ASA tools can also be used by developers in the unit-testing phase of a project. Some ASA tools can be seamlessly added as a tool bar of integrated development environments (IDEs) such as Eclipse and IBM Rational Application Developer. At the time of this writing, there were seventeen open source ASA tools (primarily lint types) and thirty-eight commercial ASA tools listed on wikipedia.org.

The ASA tool used in our case studies classifies defects in four major categories:

- Defects
- Header file problems
- Low-level interface problems
- Security vulnerabilities

At the time of the case studies, the tool was capable of detecting more than fifty-two defect types from coding style, memory management, and null pointer dereferences to weak cryptography and access problems. Depending on the application, some defect types will be more critical than others. The ASA tool Klocwork provides a mechanism for tailoring the defect types detected at build time. In fact, Klocwork also provides an application-programming interface that allows an organization to create highly customized code checkers. The case studies presented here focus only on the defect checkers that are shipped with the tool and should be adequate for the vast majority of applications.

The types of defects detected are:

- Coding style
- Concurrency
- Memory-management problems
- Null pointer dereference
- Use of uninitialized data

There are eleven sub-types of defects found under coding style:

- Assignment in condition
- Inappropriate iterator usage
- Inconsistent use of types
- Invalid pointer arithmetic
- Loss of data

Memory Leak (mlk.must)

```

1 class A {
2     void foo();
3 };
4 void A::foo()
5 {
6     int *ptr = new int;
7     *ptr = 25;
8     ptr = new int;
9     *ptr = 35;
10 }

```

Klocwork produces a defect report like the following:

mlk.must.cc:8:Error:Memory leak. Dynamic memory stored in 'ptr' allocated through function 'new' at line 6 is lost at line 8
mlk.must.cc:10:Error:Memory leak. Dynamic memory stored in 'ptr'

Figure 1

Memory Leak (mlk.might)

```

1 void foobar(int i)
2 {
3     char *p = (char*) malloc(12);
4     if(i) {
5         p = NULL;
6     }
7     return;
8 }

```

Klocwork produces a defect report like the following:

mlk.might.c:7:Error:Possible memory leak. Dynamic memory stored in 'p'

Figure 2

- Statement has no effect
- Suspicious semicolon
- Unreachable code
- Unused code
- Unused data
- Suspicious return values

There are seven sub-types of defects under memory management:

- Attempt to use memory after free
- Freeing mismatched memory
- Freeing non-heap memory
- Freeing unallocated memory
- Inconsistent freeing of memory
- Memory leak
- Returning reference to local variable

The types of header file problems detected are:

- Cycle in include files
- Missing include files
- Unnecessary include files
- Missing direct include files

The types of low-level interface problems detected are:

- Object defined in header and declared in header
- External object defined in header
- Static object defined in header
- Duplicated header
- File uses local declaration of object without using interface file
- Usage of object without declaration

NULL Pointer Dereference Example:

```
1 void npd_gen_must() {
2     int *p = 0;
3     *p = 1;
4 }
```

Klocwork produces a defect report like the following:

npd.gen.must-ret-expl.c:3:3: Error(1):NPD.GEN.MUST: Null pointer 'p' that comes from line 2 will be dereferenced at line 3

Figure 3

Might Be a Dereference Example:

```
1 void npd_gen_might(int flag, char *arg) {
2     char *p = arg;
3     if (flag) p = getNull();
4     if (arg) {p = arg;}
5     xstrcpy(p, "Hello");
6 }
7     Void xstrcpy(char *dst, char *src) {
8     if (!src) return;
9     dst[0] = src[0];
10 }
```

Klocwork produces a defect report like the following:

npd.gen.might-ret-call.c:5:8: Error(1):NPD.GEN.MIGHT: Null pointer 'p' that comes from call to function 'getNull' at line 3 may be dereferenced by passing argument 1 to function 'xstrcpy' at line 5.

Figure 4

Array Bounds Overflow Example:

```
1 int main() {
2     char fixed_buf[10];
3     sprintf(fixed_buf, "Very long format string\n"); return 0;
4 }
```

Klocwork produces a defect report like the following:

4:Critical:Buffer overflow, array index of 'fixed_buf' may be outside the bounds. Array 'fixed_buf' of size 10 declared at line 3 may use index values 0..24

Figure 5

- Multiple declarations for object
- Multiple interface files
- Multi-kind object definitions
- Missing some interfaces
- Only declaration found for object
- Only definition found for object
- Global object used locally only
- Buffer overflow
- DNS spoofing
- Ignored return values
- Injection flaws
- Insecure storage
- Unvalidated user input

Access problems consist of three subtypes:

- Improper sequencing
- Least privilege
- Time of creation—time of usage

The types of security vulnerabilities detected are:

- Access problems

Insecure storage consists of two subtypes:

- Poor randomization
- Weak cryptography

Presented next are example code snippets in C/C++ and the text messages generated by Klocwork, which are examples of defects that fall into one of the four major categories. These examples are taken from the Klocwork user guide.

Figure 1 shows a memory management problem. Note in figure 1 that Klocwork has codes for each defect and has the concept of must and might. Here, memory is allocated on line 6 with the variable `ptr` and allocates memory again on line 8 with the same variable without freeing the memory first, which causes a memory leak. The method returns at line 10 without freeing the memory a second time, and since the memory was allocated within a method block, the method return goes out of scope and creates a second memory leak.

Figure 2 is an example of the might condition for the memory leak defect. Memory is allocated on line 3 and set to NULL only if the variable `i` is not zero. If `i` is zero, line 7 returns without freeing the memory.

On line 2 in figure 3, a pointer to an integer is declared and set to NULL. Then, on line 3, the pointer is dereferenced. The message at the bottom is straightforward.

In figure 4, if `arg` is NULL, a NULL pointer will be passed to the function `xstrcpy()`, which will dereference it, causing a crash. According to Klocwork's documentation, "This example illustrates that a pointer value that can come either from a local assignment of a NULL constant or from a call to a function that will return NULL, might be dereferenced either explicitly or through a call to a function that will dereference it without checking for NULL."

Notice that the message starts with `Error(1):NPD.GEN.MIGHT`. For each defect, Klocwork assigns a severity level. "Error" is one of those severity levels.

On line 2 of figure 5 an array of characters of size ten is declared, and on line 3 a string of size twenty-four is copied into the `fixed_buf` array, thereby overflowing the array bounds and causing

LOC_METHOD	FUNCTION	Value	process_result_set
1	LOC_METHOD	Lines_of_code	27
2	NCNBLOC_METHOD	Non_comment_non_blank_source_lines	27
3	NOOPUSED	Number_of_operands_used	48
4	NOODISOPUSED	Number_of_distinct_operands_used	22
5	NOOPRUSED	Number_of_operators_used	41
6	NOODISOPRUSED	Number_of_distinct_operators_used	11
7	NOOPARMS	Number_of_parameters	2
8	NOCALLS	Number_of_calls	7
9	NOCALLSOC	Number_of_calls_to_methods_from_other_classes	9
10	NOPAROTHER	Number_of_parameters_passed_to_functions	16
11	NOEXSTAT	Number_of_expression_statements	6
12	NOSTAT	Number_of_statements	12
13	NOLoops	Number_of_loops	2
14	NOIF	Number_of_conditional_statements	2
15	NOBRANCH	Number_of_branches	1
16	MAXLEVEL	Maximum_level_of_control_nesting	4
17	AVERLEVEL	Average_level_of_control_nesting	2

Figure 6

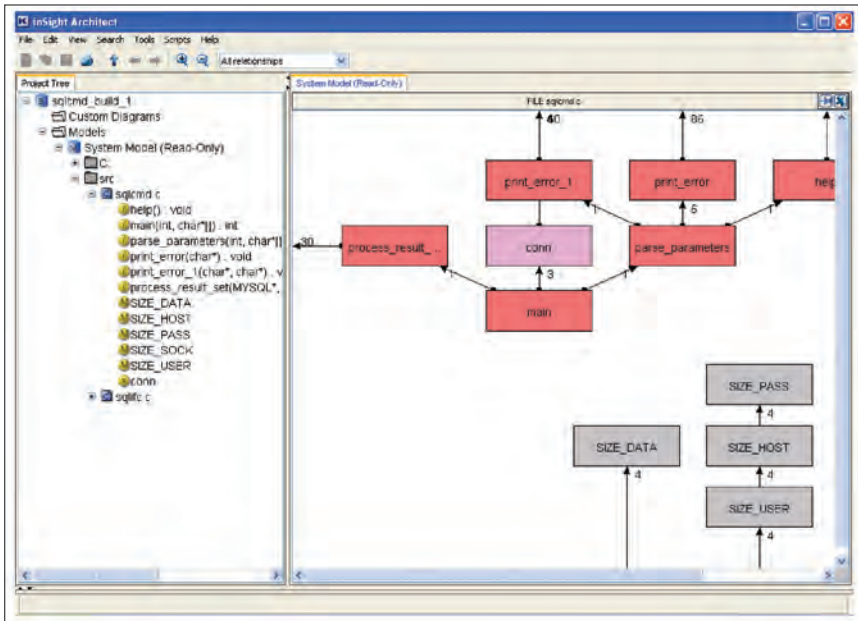


Figure 7

unpredictable and non-deterministic behavior of the program. Note the message: The severity level of this defect is “Critical,” the highest level.

In addition to finding defects, the ASA tool allows users to categorize and prioritize defects found. The tool also is smart enough to recognize when defects have been fixed and that other unfixed, detected defects have shifted position in the source code. This capability is because the tool keeps track of defects by attributes and characteristics, not simply by location. The tool has a rich set of metrics, which includes cyclomatic complexity, detected fault density, and more than one hundred others, as shown in figure 6, a portion of a metrics report.

The ASA tool also contains an architectural tool that allows the source

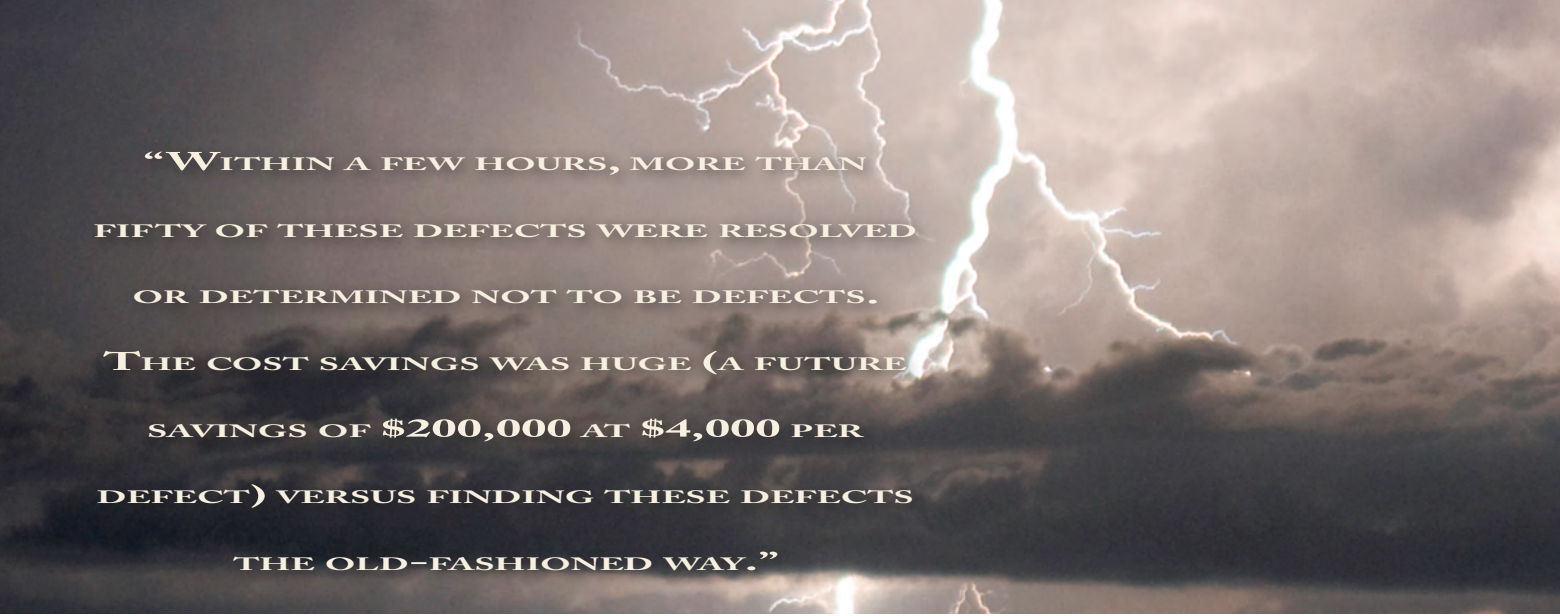
code to be displayed in graphical format. The graphs show cyclic relationships between functions or methods within files and allow the user to drag and drop the methods or functions into different files to resolve the cyclic relationships. The tool also suggests the source code changes required to re-architect the source code, as shown in the Architect tool in figure 7.

So what are some of the benefits of using ASA tools? To answer that question, consider the time it takes a developer debugging code to discover and fix software defects. Two case studies are presented. The first case study is scientific research code written in C++. This is the environment and toolkit used primarily by physicists to create highly portable 2D and 3D simulations. These pro-

grams change frequently and are highly algorithmic and complex. They also run in parallel on some of the world’s largest and fastest computers. The second case study is C++ code used in security software products. This code has been stable for years with occasional functionality upgrades. These case studies represent two very different types of software.

Our first case study was done using a physics simulation application written in the C++ programming language that is used by other application development teams at Lawrence Livermore National Laboratory (LLNL), which uses two of the better performing ASA tools, Klocwork K7 and Coverity Prevent. The first case study presented is for the Klocwork K7 tool. The application in this case study involved approximately 137,000 lines of executable C++ code, and approximately 1,000 suspected defects were discovered by Klocwork. The case study focused on just one defect (buffer overflow) of which eighty-two instances were discovered. This defect type was chosen because the behavior of the code at run time is unpredictable, it isn’t repeatable from run to run, it typically is not caught by debuggers, and it is one of the most time-consuming defects to isolate and repair. The value of the ASA tool became apparent when it took fewer than thirty minutes to run the static analysis on the application, and all eighty-two detected buffer overflows were fixed by the developer in just a couple of days. This translates into customers’ not having to deal with intermittent and non-repeatable reliability problems and your not having to respond to as many customer complaints and release cycles with fixes, which saves money.

The second case study was conducted on 345,000 lines of code written by laboratory developers in Borland C++ Version 6. This software was used for user interfaces and gateways for a physical security system. The C++ code has been running in a 24/7 environment for four years and contains a variety of applications such as Alarm Display and Configuration Editor. This software runs at multiple sites, and the code is very stable with rare, unexplained errors. Even though the level of quality of this software was generally acceptable, our case



“WITHIN A FEW HOURS, MORE THAN FIFTY OF THESE DEFECTS WERE RESOLVED OR DETERMINED NOT TO BE DEFECTS. THE COST SAVINGS WAS HUGE (A FUTURE SAVINGS OF \$200,000 AT \$4,000 PER DEFECT) VERSUS FINDING THESE DEFECTS THE OLD-FASHIONED WAY.”

study attempted to determine if the tool was capable of finding the source of the rare, unexplained errors, since the consequence of these occasional errors could be large:

only be asked to repair suspected defects that have been determined to be real. The time to prefilter—four hours for 200,000 lines of code—is still relatively small when compared to typical peer

traditional environments and mission-critical applications.

Contemporary ASA tools have proven to have three major advantages on our software projects:

- They are capable of finding about 50 percent of all possible defect types quickly, saving time and reducing development and testing costs.
- They require testers to understand computer languages.
- Prefiltering is the most time-consuming task.

The ASA tool found 285 potential defects:
Critical: 56 (Null pointer dereference, buffer overflows)
Severe: 14 (Use of free memory)
Error: 193 (Memory leaks and uninitialized variables)
Warning: 21 (Inconsistent case labels)
Other: 1

Within a few hours, more than fifty of these defects were resolved or determined not to be defects. The cost savings was huge (a future savings of \$200,000 at \$4,000 per defect) versus finding these defects the old-fashioned way. It is expected that all 285 bugs could be fixed with forty hours of developer time and that traditional methods would have taken weeks if not months to fix.

Using an ASA tool requires access to the source code, external libraries, and any make and build scripts. It usually takes an hour or two to actually build the code. Prefiltering the code reports after running the build is by far the most time-consuming task. Typical prefiltering times run four hours for every 200 defects. For code with a defect rate of five per 1,000 lines, this would mean about four hours to prefilter 200,000 lines of code. Each suspected error must be categorized as either fix, not a problem, or questionable. Early experiences in handing back code directly to developers before prefiltering were unsatisfactory. Prefiltering assures that a developer will

inspection times of an hour or so to inspect 300 lines of code.

LLNL has analyzed more than 1.9 million lines of C, C++, and Java source code using Klocwork's K7. To date, more than 3,000 defects or security vulnerabilities have been discovered and repaired. It is costing about \$100 a defect to find defects using an ASA tool, and it is estimated that a defect would cost \$4,000 if found by users after release. The bottom line is that ASA technology in the higher end tools such as Coverity Prevent and Klocwork K7 works, and the return on investment easily justifies the tool. The only downside is that some prefiltering time is necessary to eliminate the 15 to 20 percent false-positive rate, the skill set of tool users requires an understanding of the source language being analyzed, and not all of the defects found are of the show stopper variety. ASA tools support the contemporary agile approaches to software development, as these tools can be run many times in a thirty-day sprint or timebox. They also can be used in

Our future plans include insertion of static-analysis capability early in the development lifecycle during the code and debugging phases by integrating the tool into the developer's IDE. Additional research is being conducted at LLNL on how to fix the code automatically after a defect is discovered. Because developers can see an immediate benefit of ASA tools, acceptance has been rapid. In fact, at LLNL, software quality engineering organizations are finding that developers actually are seeking out the tool once they understand what it can do for them. An ASA tool can have no better endorsement than that. **{end}**

LLNL-JRNL-402003-DRAFT

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

REFERENCES:

- 1] Boris Beizer, *Software Testing Techniques*, Second Edition, Van Nostrand Reinhold, page 57, Table 2.1.