



# Satisfiability: A new generation of static analysis

By Ben Chelf

*Source code analysis has had a mostly deserved bad reputation in software development because analyses often took too much time or produced excessive noise and a large percentage of bogus results (false positives). With low signal-to-noise ratios, most source code analysis technologies and products quickly became shelfware after a few uses. The promise of new static analysis solutions is tantalizing for developers because it offers the ability to find bugs before software is run, improving code quality and dramatically accelerating the availability of new applications. Though static analysis has historically struggled to deliver on this promise, a groundbreaking technique applied in the static analysis field may help fulfill its potential.*

The software development challenges of today require companies to look for innovative ways to remove the critical flaws and vulnerabilities in software before it gets to the quality assurance lab or, worse, out into the field (see sidebar). Fortunately for developers, source code analysis is now up to the challenge. By combining breakthrough techniques in the application of Boolean satisfiability to software with the latest advances in dataflow analysis, the most sophisticated source code analysis can boast out-of-the-box false positive rates as low as 10 percent and scalability to tens of millions of lines of C/C++ or Java code.

## ***The price of failure in software***

A 2002 National Institute of Standards and Technology (NIST) study titled “The Economic Impacts of Inadequate Infrastructure for Software Testing” estimated that software errors cost the U.S. economy an estimated \$59.5 billion annually. The report states that leveraging test infrastructures that allow developers to identify and fix defects earlier and more effectively could eliminate more than one-third of these costs.

For more information on this study, visit  
[www.nist.gov/public\\_affairs/releases/n02-10.htm](http://www.nist.gov/public_affairs/releases/n02-10.htm).

Software developers can use static analysis to automatically uncover errors that are typically missed by unit testing, system testing, quality assurance, and manual code reviews. By quickly finding and fixing these hard-to-find defects at the earliest stage in the software development life cycle, organizations are saving millions of dollars in associated costs.

## Software DNA Map: The foundation of superior code analysis

If developers want superior analysis, they must have a perfect picture of the software because their analysis will only be as accurate as the data it is based upon.

Imagine someone gave a software development team a system of a few million lines of code – a system they'd been working on for a long time – and asked them to draw a map of the software system. To be useful, this map would need to address all the following details: how every single file was compiled for all targets, how each set of files was linked together, the different binaries that were generated, the functions within each file and the corresponding callgraph, all the different control flow graphs through each function, and on and on. If the developers attempted to do this by hand, it would be practically impossible.

Today, automating this onerous task is possible, and the process of creating such a picture, known as a *Software DNA Map*, opens the door for static analysis to significantly improve code quality and security. A Software DNA Map is an extremely accurate representation of a software system based on understanding all operations that the build system performs as well as an authentic compilation of every source file in that build system. By providing an accurate representation of an application to the function, statement, and even expression level, the Software DNA Map enables static code analysis to overcome its previous limitations of excessive false positives and deliver accurate results that developers can put to immediate use.

## Static code analysis primed for advancement

Combining breakthroughs in dataflow analysis with a Software DNA Map has created substantial benefits for development organizations by enabling them to detect defects early in development. However, the performance and accuracy of these analysis solutions can still be improved. A groundbreaking new technology called *Boolean satisfiability* is poised to greatly expand static code analysis capabilities.

### Boolean satisfiability

In complexity theory, the Boolean satisfiability problem (SAT) is a decision problem whose instance is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: Given the expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true? A formula of propositional logic is said to be satisfiable if logical values can be assigned to its variables in a way that makes the formula true.

The concept of Boolean satisfiability is not a new one, but recently, efficient SAT solvers have been developed to solve very complicated formulas with millions of variables. While used heavily in the electronic design automation industry to aid in chip design verification technologies, the application of SAT solving to software analysis has remained untouched.

### SAT solver

A SAT solver is a computer program that takes in a formula of variables under the operations of AND, OR, and NOT and determines if there is a mapping of each individual variable to true and false such that the entire formula evaluates to TRUE (satisfiable). If no such assignment exists, the SAT solver indicates that the formula is unsatisfiable and provides a proof demonstrating that it is unsatisfiable.

The application of SAT to software requires that source code be represented in a form that can be plugged automatically into a SAT solver. Fortunately, with a Software DNA Map, all the necessary information from the code is available to transform it into any representation desired. Because SAT solvers deal in TRUE, FALSE, AND, OR, and NOT, the relevant portions of this program can be transformed into these constructs. Take an 8-bit variable as an example:

```
char a;
```

To represent 'a' as TRUES and FALSES, those 8 bits (1s and 0s) can be each thought of as TRUES and FALSES, so 'a' becomes an array of 8 Boolean values:

```
a0, a1, a2, a3, a4, a5, a6, a7
```

The operations that make up expressions in the code also must be translated. All expressions in code can be converted to an equivalent formula of AND, OR, and NOT. The thinking behind this is that a compiler must turn these operations into instructions in machine code and that machine code must run on a chip. The chip's circuitry is nothing more than pushing 1s and 0s (high voltage and low voltage) through a number of gates (all of which can be simplified to AND, OR, and NOT); therefore, this indicates that such a mapping exists. For example, to convert the expression:

```
a == 19
```

into a formula, the following expression would do the trick:

```
!a0 ^ !a1 ^ !a2 ^ a3 ^ !a4 ^ !a5 ^ a6 ^ a7
```

In this example, a<sub>0</sub> is the high bit of 'a' and a<sub>7</sub> is the low bit. Plugging this into a SAT solver would render the following assignment of variables for the formula to be satisfied:

```
a0 = False (0)  
a1 = False (0)  
a2 = False (0)  
a3 = True (1)  
a4 = False (0)  
a5 = False (0)  
a6 = True (1)  
a7 = True (1)
```

Taking that, as binary 00010011, shows that it is equivalent to 19.

Once the entire software system is represented in this format of TRUES, FALSES, NOTS, ANDS, and ORS, a wide variety of formulas can be constructed from this re-presentation and SAT solvers can be applied to analyze the code for additional, more sophisticated quality and security problems. It is this *bit-accurate representation* of the software that enables more precise static analysis than previously was possible based solely on dataflow techniques.

## Writing better code

One early application of SAT solvers for static analysis is false path pruning. When performing dataflow analysis, sometimes a defect will be reported on a path that is infeasible (or unsatisfiable) at runtime. This reported defect should be

eliminated from the static analysis results since there is no possible combination of variables where it could actually happen in the software system. The methodologies for dealing with this problem within the dataflow framework pale in comparison to the abilities a bit-accurate representation and a good SAT solver can provide in this regard.

For example, with this bit-accurate representation of the source code, developers can construct a formula that is a conjunction (AND) of all the conditions in a path that lead to any given defect discovered by dataflow analysis. By solving this formula, the SAT solver indicates if there is a set of values for the variables involved in all the conditions such that the path can actually be executed. If the SAT solver says “satisfiable,” then the path is feasible at runtime. If the SAT solver says “unsatisfiable,” the path can never be executed and no bug should be reported.

Consider the earlier discussion of static analysis history and the problems with excessive false positive results. By identifying false paths with SAT, developers can prune them from their static analysis results. This enables them to focus testing and analysis efforts on potential problems that have a real possibility of compromising the project at hand.

It’s important to note that false path pruning is just one example of how SAT can be leveraged to provide more accurate source code analysis. Other potential applications of SAT include the ability to find problems such as string overflows, integer overflows or deadcode, and the use of assertion-based checking to identify difficult-to-find logic bugs (see sidebar). While some instances of the defects in these categories can be discovered today, SAT-based analysis allows developers to build on the success of existing dataflow analysis to reach new levels of accuracy and precision in static code analysis.

SAT will benefit software developers and architects by fundamentally changing the way they view static analysis of their source code. Just as going from simple “grep” parsing through code to practical dataflow analysis earlier this decade was a huge eye-opener for many developers (“You mean, it can show me the whole path to get to a real bug?”), leveraging SAT solvers for static analysis can impress anyone who writes software (“How could you possibly know that about my code?”). Because of this, SAT may be the breakthrough that enables static analysis to deliver on more of its long-awaited promise.

## Leveraging SAT techniques

For years, embedded software developers have looked for products that could auto-matically and effectively find software defects early in the development cycle. Fixing software bugs early can dramatically reduce the time it takes to bring a software product to market and potentially save millions of dollars in costly product recalls. However, tools for discovering defects automatically have had many false starts because of their failure to grasp a complete picture of the software. Older technologies incorrectly identified harmless code inconsistencies as defects and wasted developers’ time.

By providing complete understanding of a given build environment and source code, a Software DNA Map can allow organizations to leverage existing SAT techniques to produce the highest-quality software possible. Leveraging SAT within static code analysis, companies can immediately find and fix software defects in source code at the beginning of the development cycle, thereby ensuring product quality and accelerating the availability of new applications to bring to market. **ECD**

***Ben Chelf** is CTO of San Francisco-based Coverity, Inc. Before he cofounded Coverity, Ben was a founding member of the Stanford Computer Science Laboratory team that architected and developed Coverity’s underlying technology. He is one of the world’s leading experts on the commercial use of static source code analysis. Ben holds BS and MS degrees in Computer Science from Stanford University.*



**Coverity, Inc.**  
415-321-5200  
bchelf@coverity.com  
www.coverity.com

## Potential uses of SAT in static analysis

### False path pruning

A bit-accurate solution to the formula of the conjunction of all conditions on a path where a defect is about to be reported eliminates many false positives that plagued dataflow analysis solutions for years.

### Integer overflow

Using a SAT solver to check key arithmetic operations (in-loop bounds, calls to memory allocation functions, as an index into an array, and so on) to verify that unexpected overflow conditions do not occur in the computation.

### Symbolic checkers

Superior defect detection capabilities for the following classes of defects:

- Buffer overflow detection
- String overflows
- Deadcode

### Assertion-based checking

- Software assertions can be complex and hard to check, and currently are only leveraged at runtime
- Vision: ability to write assertions using a well-known primitive library
  - No need to learn an esoteric formal language
  - Restrict assertions to checkable properties
- Result: developers will be able to find some program logic bugs
  - This is an area static checkers today simply ignore