Detecting Domain-specific Coding Errors with Static Analysis

Paul Anderson, Ph.D.
VP of Engineering, GrammaTech

Abstract

Advanced static-analysis tools for finding programming errors have become very popular recently. These detect many common generic programming errors such as null pointer dereferences, buffer overruns, and race conditions. Most major static-analysis tools also provide an interface that can be used to find domain-specific errors. This paper describes how custom domain-specific checkers can be used to improve software quality in complex embedded systems.

1. Introduction

Static analysis is a term that describes techniques that compute run-time properties of programs, without actually executing them. Static-analysis tools are typically used to find program defects. The first generation of static-analysis tools, exemplified by the *lint* family of tools, had limited capability and were only capable of finding superficial defects. The latest generation, such as GrammaTech's *CodeSonar*® use highly sophisticated whole-program analyses to find deep semantic problems [1]. This paper is about this class of tools, and refers to them as *advanced* static-analysis tools.

Out of the box, advanced static analysis tools are good at finding programming defects that arise because a fundamental rule of the languages is being violated (e.g., a buffer overrun), or because an API is being used incorrectly (such as closing the same file descriptor twice). Many tools are also capable of finding violations of stylistic rules (e.g., don't use *goto*). These tools have proven to be effective at finding serious problems, and they have been widely adopted. For example, see [2, 3].

An often under-appreciated aspect of these tools is that they are extensible. They can be configured or programmed to find violations of domain-specific rules too. So if you have rules for using an internal API, or require programmers to use a particular idiom, then it is often possible to write a checker that signals violations of those rules. Programmers can, often with a little programming effort, dramatically increase the value they get from the tools.

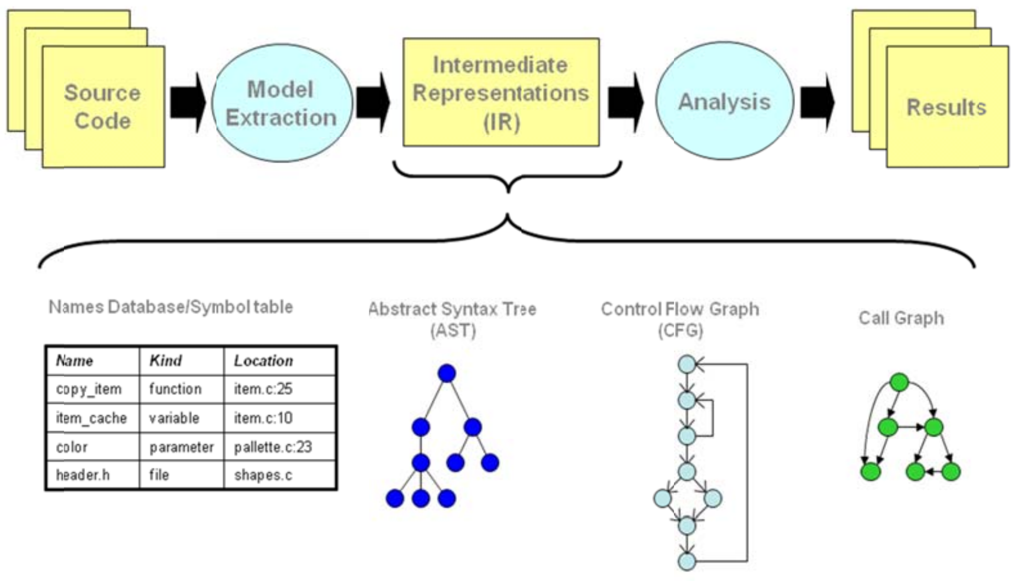
This paper describes some of the ways in which static-analysis tools can be extended. Section 2 describes how these tools work. Section 3 uses examples to describe several ways in which the tools can be customized. Section 4 concludes with some recommendations.

2. How Advanced Static Analysis Works

In order to understand the different ways in which extensions can be written, it first helps to know what static analyis is, and how these tools work.

Static-analysis tools work very much like compilers. They take source code as input, which they then parse and convert to an intermediate representation (IR). The IR typically comprises the program's abstract syntax tree (AST), the symbol table, the control-flow graph (CFG), and the call graph. A block diagram of the architecture of an advanced static-analysis tool is shown below in Figure 1. Whereas a compiler would use the IR to generate object code, static-analysis tools retain the IR, and checkers are usually implemented by traversing or querying the IR looking for particular properties or patterns that indicate defects. A simple checker, such as one that looks for simple syntactic properties (e.g., *goto* statements), would search the abstract syntax tree for constructs that match that pattern. A more complex checker might examine the call graph or CFG.

The advanced tools get their power from sophisticated symbolic execution techniques that explore paths through the control-flow graph. These algorithms keep track of the abstract state of the program and know how to use that state to exclude consideration of infeasible paths. This level of complexity is required in order to find the serious bugs while keeping the level of false positives low.


Source Code
Model Extraction
Intermediate Representations (IR)
Analysis
Results
Names Database/Symbol table
Abstract Syntax Tree (AST)
Control Flow Graph (CFG)
Call Graph

Name	*Kind*	*Location*
copy_item	function	item.c:25
item_cache	variable	item.c:10
color	parameter	pallette.c:23
header.h	file	shapes.c

Figure 1. The architecture of an advanced static-analysis tool.

Although all advanced static-analysis tools have an architecture much like the one shown, the details differ somewhat. In *GrammaTech CodeSonar®*, all of the intermediate representations are retained, and an interface allows end users to author code that inspects these representations in various ways. The techniques with which new checkers can be written are described in the next section.

3. Custom Checkers

No two tools provide exactly the same interface or techniques for implementing custom checkers, but three mechanisms are common, and are described in more detail in the subsequent subsections:

- Existing checkers can be extended by adding directives to a configuration file.
- The user can add annotations to their code that instruct the analysis to look for certain properties. In *CodeSonar®*, these annotations can be done on the side in an aspect-oriented fashion if users do not wish to perturb the source code.
- An API allows users access to all of the intermediate representations. Typically, a visitor pattern is used that allows extensions to piggyback on traversals the analysis is already doing.

Although most tools have similar mechanisms, the examples are shown for *CodeSonar*.

3.1. Configuration Files

These advanced static analysis tools implement dozens of checkers. Often, a user needs a checker that is only slightly different than a built-in one, and many of them have been designed to be extensible. One class of checkers finds functions whose use is forbidden. For example, the C library function *gets* is notoriously insecure (and is now officially deprecated). The checker is implemented by a phase that finds references to function names, and then matches them against a set of regular expressions. It is a simple matter to add additional regular expressions to this set by adding lines to a configuration file.

A similar process applies to extend the set of functions whose return value should always be checked, or to specify which functions take format string parameters.

3.2. Code Annotations

The second way in to write checkers is to add annotations to the code.

Suppose for example that there is an internal function named *foo* that takes a single integer parameter, and that it is dangerous for that parameter to have the value -1. A check for this case could be implemented by adding some code to the body of *foo* as follows:

```
void foo(int x)
{
#ifdef __CSURF__
    csonar_trigger(x, "==", -1,
                   "Dangerous call to foo()");
#endif __CSURF__
    …
}
```

The *#ifdef* construct ensures that this new code is not seen by the regular compiler. However when this code is analyzed by *CodeSonar*, the call to *csonar_trigger* is seen. Thus this call is never actually executed, but is instead interpreted by the tool as a directive to the underlying analysis engine. If the analysis considers that the trigger condition may be satisifed, then it will issue a warning with the given message.

Of course in most cases it is not appropriate to require that programmers interleave these annotations with the code, so there is an alternative way to implement this kind of check that allows it to be written in a separate file, which avoids the need for the code to be edited. This approach is also appropriate when the source code for *foo* is not available, such as when it is in a third-party library. To do this, the author of the checker would write a *replacement* function as follows:

```
void csonar_replace_foo(int x)
{
    csonar_trigger(x, "==", -1,
                   "Dangerous call to foo()");
    foo();
}
```

When the analysis sees the definition of *csonar_replace_foo*, it treats all calls in the code to *foo* (except the one inside *csonar_replace_foo*) as if they were calls to *csonar_replace_foo* instead.

This trigger idiom is good for checking temporal properties, particularly sequences of function calls. Suppose there is a rule that says that *bar* should never be called while *foo* is executing. A check might be implemented as follows:

```
static int foo_is_executing = 0;
void csonar_replace_foo(int x) {
    foo_is_executing = 1;
    foo();
    foo_is_executing = 0;
}
```

```
void csonar_replace_bar(void) {
    csonar_trigger(foo_is_executing, "==", 1,
                   "Call to bar from foo");
    bar();
}
```

Note that a global state variable is used to record whether or not *foo* is active. Before entry to *foo* it is set to one, and then reset to zero after *foo* returns. This variable is then checked in the trigger in *bar*, and if set to one, a warning will be issued.

The example above shows how a global property might be checked. The same mechanism can be used to write checks for properties of objects. For example, it is illegal to read from a file object after it has been closed. In such a case the state being checked must be associated with the file object. The extension API allows users to specify *attributes* that can be attached to objects. These can be thought of as state variables that can be associated with objects. In the example just given, an attribute can be used to encode whether a file object is open or closed, and the checker for a reader function can test the value of that attribute and issue a warning if the file is in fact closed.

This approach allows users to author static checks almost as if they were writing dynamic checks. The API for this kind of check is small, and the language is regular C, so there is a shallow learning curve. This simplicity is deceptive — the technique can be used to implement fairly sophisticated checks. In *CodeSonar*, many of the out-of-the-box checks are implemented this way.

3.3. API for Intermediate Representations

The final way to implement a custom check is to use the API that gives access to the underlying IR. This technique has been used by *CodeSonar* users to implement a variety of checkers. For example, one company makes a highly sophisticated electronics manufacturing system, controlled by in excess of a million lines of code. They employ a custom idiom for handling hardware errors. If this is not followed consistently, then it can lead to excessive expensive downtime for their customers. They created a custom checker in *CodeSonar* that finds locations in the code where the idiom is incorrect.

This API can be used for other program analysis tasks too. A medical device company uses *CodeSonar* to identify potential tasking problems in their code, and to emit information that allows them to interactively explore properties of stack configurations.

Many checks can be written using a visitor pattern. A visitor is a function that is invoked on every IR element of the appropriate type; there are different visitor types for different IR constructs. When they are present, visitors piggyback on the various traversals carried out by the analysis. Visitors can be defined for files, identifiers, subprograms, and nodes in the control-flow graph (which correspond roughly to program statements), and the syntax tree. The interface to a visitor allows for pattern matching against the construct. This way a checker author can easily search for constructs without having to know exactly how they are represented.

For example, suppose there is a rule that no variable is allowed to contain upper-case characters. The checker for this would be best implemented by writing a function that takes an identifier as input, checks that it represents a variable, scans it for upper-case letters, and issues a warning if one is found. This function would be registered as a visitor for the table of identifiers.

CodeSonar has an additional kind of visitor, which is invoked during the path exploration of the control-flow graph. At each step along the path, the check can query the abstract state of the program, so the implementation can ask questions such as "what is the value of this variable at this point". This allows the checker author to write sophisticated checks that leverage the built-in program analysis capabilities of the tool. A key aspect of this kind of checker is that it uses the part of the analysis that eliminates the exploration of infeasible paths, which automatically reduces the probability of false positive results.

4. Recommendations

Advanced static-analysis tools have become essential tools for software developers because they have proven effective at finding serious flaws. Users of such tools should consider writing custom checkers in order to dramatically increase their return on investment.

References

1. GrammaTech Inc., CodeSonar, http://www.grammatech.com/products/codesonar/.

2. Jetley, R. P., Anderson, P., and Jones, P. L., *Static Analysis of Medical Device Software using CodeSonar*. In *Static Analysis Workshop (SAW)*. 2008. Tucson, AZ: ACM Press.

3. Pope,G., Ferrari,K., and Oliver,B., *Give Your Defects Some Static* -- *Using Static Analyzers to Debug Your Code*. in *Better Software*. July, 2008. pp. 36-42.

GRAMMATECH

GrammaTech, Inc.
531 Esty Street
Ithaca, NY 14850
Tel: 607.273.7340
www.grammatech.com