CodeAdvisor: Rule-Based C++ Defect Detection Using a Static Database

C++ SoftBench CodeAdvisor is an automated error detection tool for the C++ language. It uses detailed semantic information available in the SoftBench static database to detect high-level problems not typically found by compilers. This paper describes CodeAdvisor and identifies the advantages of static over run-time error checking.

by Timothy J. Duesing and John R. Diamant

C++ is a powerful successor to the C language that has all of C's features plus a lot more, including constructors, destructors, function overloading, references, inlines, and others. With this added power come more options to manage, more ways to do things right, and inevitably, more ways to go wrong. C++ compilers can find syntactical errors, but they do not find errors involving constructs that are legal yet unlikely to be what the programmer intended. Often, problems of this nature are left to be found during testing or by the end user. Attempts to find these defects at an earlier and less expensive stage of development sometimes take the form of code inspections or walkthroughs. While careful walkthroughs can find some of these errors, formal inspections are time-consuming and so expensive that they are usually only applied to small pieces of the code.

Since C++'s introduction in the early 1980s, a large body of experience with the language has accumulated and many works have appeared that describe common pitfalls in the language and how to avoid them.[1-5] While some of these problems can be quite subtle, some of them are also straightforward enough that a program can be created to detect them automatically,[6] as long as that program can be supplied with sufficiently detailed information about the code's structure. The SoftBench static database (see ***Article 3***), with its semantic information, provides an opportunity to create a tool that can do just that. This article is about such a tool: C++ SoftBench CodeAdvisor.

CodeAdvisor: An Automated Rule Checker

CodeAdvisor distills its knowledge of what are likely to be coding errors as a set of rules that alert the user to problems such as calling virtual functions from constructors, mixing iostream routines with stdio routines, local variables hiding data members, and so on. Each rule is a set of instructions that queries the static database for the information of interest and then performs the logic to test whether that potential error condition is present. When it detects a rule violation, CodeAdvisor displays the violation's location (file, line number) in an error browser that lets the user navigate quickly and easily to the problem site and use an editor to correct it. Online help is available to present more explanation of the violation, possible ways to correct the problem, references for further information, and when appropriate, exceptions to the rule.

CodeAdvisor detects rule violations by performing static analysis of the code using the SoftBench static database. Static analysis differs from the dynamic or run-time analysis done by debuggers, branch analyzers, and some performance tools in that all of the available code is examined. Dynamic analysis examines only code that is actually executed and cannot find defects in branches that are never taken. Also, dynamic analysis requires that the code be far enough along so that it can be actually executed. Static analysis, on the other hand, can be performed as soon as the code compiles, even if the code cannot yet successfully run.

Because it is automated, CodeAdvisor will tirelessly check all the rules it knows against all of the code. This is practical only for relatively small pieces of code during inspections done by hand. Unlike a human code reviewer, CodeAdvisor never gets so tired or bored that it misses a rule violation it's been programmed to find. While CodeAdvisor cannot replace inspections completely (there will always be problems that cannot be detected automatically), it can be a good complement to traditional code inspections, freeing developers to focus on higher-level problems by weeding out the detectable problems first.

Example Rule: Members Hidden by Local Variables or Parameters

Let's look at an example of one of the rules CodeAdvisor implements and examine how it uses the static database to find a rule violation. Consider the small program in Fig. 1. The class Vehicle with its two-line member function SetSpeed looks simple enough. The constructor for Vehicle sets the initial speed to zero, so we would expect to get a current speed of zero at the start of the program and we do. We might also expect that, after calling SetSpeed with a delta of 50, we would then get a current speed of 50. However, if we actually compile and run the program we find that we still get zero! Why? The problem is that a data member is hidden by a function parameter with the same name. In SetSpeed we've made an unlucky choice when we named the parameter speed, since there is a data member of the same name in the class Vehicle. When speed is modified in SetSpeed, the compiler modifies the parameter rather than the data member. The compiler will not complain since we have given it unambiguous instructions, which it will follow perfectly. If we had chosen any other name for our local variable, the example would work as expected.

```
#include <iostream.h>

class Vehicle {
private:
     int speed;
public:
     int CurrentSpeed() const { return speed; }
     void SetSpeed(int newspeed, int delta = 0);
     Vehicle() { speed = 0; }
};

// SetSpeed takes an absolute speed plus a
// delta. If absolute speed is zero, use
// current speed. Other parameters should be 0
// (2nd one defaults to 0)
void Vehicle::SetSpeed(int speed, int delta)
{
     if (!speed) speed = CurrentSpeed();
     speed = speed + delta;
};

main()
{
     Vehicle car;
     cout << ”Car’s initial speed = ”
          << car.CurrentSpeed()
          << endl;
     car.SetSpeed(0,50);
     cout << ”Car’s new speed = ”
          << car.CurrentSpeed()
          << endl;
}
```

Fig. 1. *An example of a CodeAdvisor rule violation: members hidden by local variables or parameters.*

Even in this simple setting, an error like this can be difficult to spot at a glance. In a more complex and perhaps more realistic situation, this problem might never be found in a code inspection. If we bury a few subtle defects like this in a few megabytes of code we might find that they won’t be found until actual execution exposes them as bugs.

Detecting an Error Using the Static Database

The problem, then, is how to find these kinds of defects before the user does. The context in which speed is used is what’s important here. Using speed as a parameter in most cases is perfectly valid. The only case we need to worry about is when a parameter or local variable is used within the scope of a member function and it has the same name as a data member of that class. This is where the static database is needed to make this kind of rule checking possible. The static database contains, among many other things, information about what objects are global and local within a scope, and it understands what objects are member functions and what the associated parameter list is.

One way to create a rule to detect this particular error is to first query the database to find all the classes in a program. Once we have all the classes, we can query the database for all the member functions of those classes. Then we can examine each function’s parameters and local variables looking for any members local to the class or inherited public or protected with the same name. If we find a match, we report a rule violation and output the file and line numbers of the offending symbols.

Of course, to make the rule robust, there are still a few little details that need to be considered in implementing the above algorithm. For instance, to be general, when we query the database for classes, we’ll want to find class templates as well, and if we find any, we’ll want to consider only the templates themselves and not their instances. Also, when we search for member functions of these classes we’ll want to skip any compiler-generated functions that the C++ compiler may have created by default. We may also want to handle the cases where a symbol hides a member function as well as a data member. All the information needed to handle these details is available in the static database.

Exceptions to the Rule

The types of problems for which CodeAdvisor is targeted are not the obvious or even the obscure abuses of the C++ language. Compilers are fully capable of finding these types of errors. Rather, CodeAdvisor attempts to identify a more subtle kind of problem that might be characterized as constructs that experience tells us are almost certainly not what the programmer intended, even though they are fully legal within the language. We must include the word “almost,” however, because occasionally some of the most unlikely constructs are in fact what the programmer intended. Deciding with certainty whether or not a suspicious construct will turn out to be a real problem may sometimes require knowledge that cannot be determined by a practical amount (or sometimes any amount!) of analysis, static or run-time.

To illustrate this, consider, for example, the CodeAdvisor rule that detects classes that are passed as a value parameter to a function. This may become a problem when the class passed is a derived class and virtual functions of that class are called within that function. This is because calls to that class’s virtual functions will call the base class’s versions, not the derived class’s versions. The above conditions are easy enough to check for with the static database, but they alone do not guarantee an error condition. If the function is never passed a derived class instance, no problem will occur. In some special cases, static analysis might be able to detect this additional condition but in other cases involving complex conditional branching, detection would be impractical or impossible. Run-time analysis also might be able to detect this condition in special cases, but in cases of less than 100% branch coverage or conditional branching determined by many combinations of possible external data, detection again would be impractical. In this particular example, CodeAdvisor will report the rule violation even with imperfect information because even when the problem only potentially exists, it can cause a serious problem for later code maintainers. Each rule, however, must be evaluated on its own merits to consider the possible nuisance of false positives.

In this sense, the rules can be regarded as heuristic—that is, good but not perfect guesses that a given piece of code is a genuine error. Fig. 2 illustrates the nature of the problem when a rule has imperfect knowledge of the code. The area where a heuristic rule is satisfied still contains cases where no real error exists. To report these cases when there is a reasonable amount of uncertainty as to their validity would be to bombard the user with unwanted “noise” that would distract from other real problems.

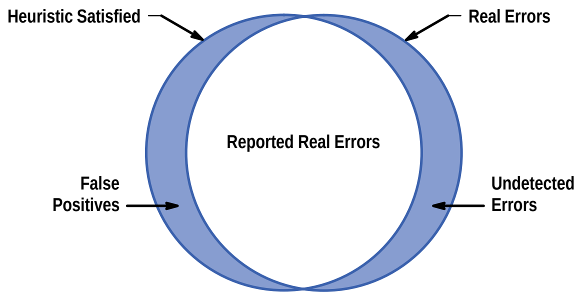

Fig. 2. *The problem of finding errors with imperfect information.*

We have reduced the noise factor in CodeAdvisor by adopting a philosophy of “no false positives” when implementing a rule. That is, when imperfect information prevents knowing with certainty if a construct causes a problem in the current setting, the code is given the benefit of the doubt unless there is also a serious potential for a future maintenance problem. In addition, for those occasional cases where a suspicious construct is reported but still deemed acceptable by the user, CodeAdvisor provides a filtering mechanism to allow the user to suppress the display of particular violations.

Summary

CodeAdvisor uses the information available in the SoftBench static database to implement a deeper level of error detection than is available with current compilers. CodeAdvisor’s static analysis has advantages over run-time analysis because all of the available code is analyzed instead of only the branches that are actually executed. An automated rule checking tool like CodeAdvisor can contribute to the code development process at an early stage, where the cost of defect repair is less expensive. CodeAdvisor complements traditional code inspection and testing, allowing developers to focus on the higher-level problems by weeding out the detectable problems first.

References

1. S. Myers, *Effective C++*, Addison-Wesley, 1992.
2. T. Cargill, *C++ Programming Style*, Addison-Wesley, 1992.
3. M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
4. Taligent Corp., *Well-Mannered Object-Oriented Design in C++*, Addison-Wesley, 1994.

5. *Programming in C++, Rules and Recommendations*, Translated from Swedish by Joseph Supanich, Ellemtel Telecommunication Systems Laboratories, 1990-1992.
6. S. Meyers and M. Lejter, "Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++," *Proceedings of the Usenix Association C++ Conference*, 1991.

▶ Go to Next Article
▶ Go to Journal Home Page