20 Experimental Program Analysis

Holger Cleve
Andreas Zeller
Lehrstuhl für Softwaretechnik, Universität des Saarlandes, Saarbrücken, Germany
{cleve,zeller}@cs.uni-sb.de

Abstract

Program analysis long has been understood as the analysis of source code alone. In the last years, researchers have also begun to exploit tests and test results. But what happens if the analysis process itself drives the test, running actual *experiments* with the analyzed program? This position paper explores some of the possibilities that arise. As a first proof of concept, our *AskIgor* web service automatically isolates cause-effect chains for given failures—without any source code: "Initially, GCC was invoked with a C program to be compiled. This program contained an addition of 1.0; this caused an addition operator in the intermediate RTL representation; this caused a cycle in the RTL tree—and this caused GCC to crash."

20.1 Reasoning about Programs

When programmers attempt to understand a program, they use a wide variety of techniques to gather knowledge:

- At the core of things lies *deduction*—reasoning from the abstract program code to what can happen in a concrete program run. Typical deduction techniques include static analysis and verification.
- Deduction, by nature, does not take concrete facts into account—that is, facts from concrete program runs. These are extracted by *observation,* as exemplified in classical debugging tools.
- If a program is executed multiple times—in a test suite, for instance—one can attempt to *induce* abstractions from the concrete runs. Techniques that exploit induction are dynamic invariants (= summarizing multiple runs), or coverage metrics (= finding code that is executed in failing runs only)
- As part of the understanding process, programmers also *experiment* with the program in question—by generating and controlling multiple runs designed to support or refute hypotheses about the program. Experimentation is hardly ever found in tools.

As sketched in Figure 20.1, these reasoning techniques form a hierarchy—for instance, induction is not possible without observation, and each technique can put to use any static knowledge as deduced from the code.

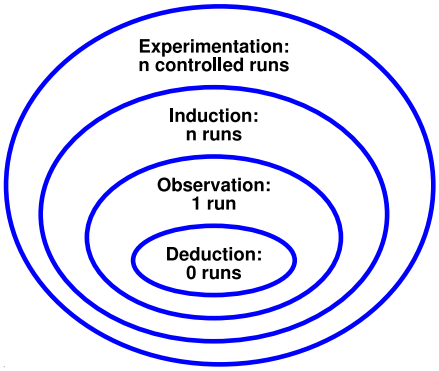

Figure 20.1: Program analysis: A hierarchy

20.2 Finding Causes

The one reasoning technique that makes use of all others is *experimentation*—the only one that can find out the *cause* of some effect. In fact, to prove causality, one needs two experiments: One where both cause and effect occur, and one where neither occur (with everything else unchanged, the cause preceding the effect, and both cause and effect being minimal). This is the way that experimental science finds the causes for natural phenomena.

Experimentation is the classical technique for *debugging:* If we want to know whether some variable x is the cause for some failure, we need to find an alternate value for x where the cause does not occur. To find such a value, and to know that changing x, instead of, say, y, is the great challenge in debugging a program—or, more generally, in understanding the cause of some effect.

One possible source for alternate values are alternate runs—that is, runs where the failure does not occur. Let us assume we have two runs: one run A where some effect occurs, and one run B where it does not. Let us further assume we interrupt program execution at some common point in the source code. If we now transfer the entire program state from A to B and resume execution, A's effect should occur in B, too. In fact, assuming deterministic execution and perfect transfer, B should behave exactly as A.

But what happens if we transfer only *part of the state* from A to B? This is an experiment whose outcome can hardly be predicted. For one thing, we would probably end up in an inconsistent state; resuming execution would lead

nowhere. However, if B now fails, one might argue that the part of the state just transferred was indeed the cause for the given effect. Using a simple strategy, this process can be repeated until the cause is narrowed down to some minimal difference between the two program states (Figure 20.2)—for instance, one pointer to the wrong element. All one needs is an automated test that checks whether the behavior of the modified run B is now A's (= the part transferred caused the effect) , still B's (= the part transferred does not cause the effect), or something different.

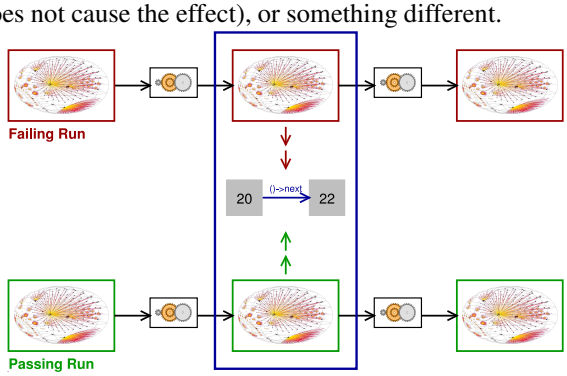

Figure 20.2: The process in a nutshell

Applying this technique at various points during the program execution eventually reveals the *cause-effect chain* from input to outcome—in the case of program failures, a short and concise diagnosis about how the failure came to be [1].

20.3 A Debugging Server

As a proof-of concept. we have built a *debugging server* called *AskIgor* ("Ask Igor")—a service that tells you why your program fails:

Submit a Program. You have a program that shows some repeatable, non-intended behaviour—for instance, the GNU compiler (GCC) crashing on some input.
You call up the *AskIgor* Web site and submit the *cc1* executable—the program that crashes. You also specify two invocations: one where the program fails, and one where it passes.

Read the Diagnosis. *AskIgor* presents the diagnosis on its Web page (Figure 20.3). The diagnosis takes the form of a *cause-effect chain*: First, this variable had this value, therefore, that variable got that value, and so on—until the program state causes the behaviour in question.
In our GCC example, the two inputs differ by the string " + 1.0" in the code (Step 1); this causes a *PLUS* operator in the intermediate RTL representation (Step 2: a new RTL node); this causes a cycle in the RTL tree (Step 3: *link* points back to itself)—and this cycle causes the compiler to crash (Step 4).

Fix the Bug. In order to fix the program, one must *break* the cause-effect chain—that is, ensure that at least one of the failure-inducing variable values no longer occurs. This is done by distinguishing *intended* from *non-intended* states—a decision left to you.
In our case, the non-intended program state is the cycle in the RTL tree. To find out how this state came to be, you can have *AskIgor* compute the cause-effect chain for the respective subsequence of the execution ("How did this happen?").

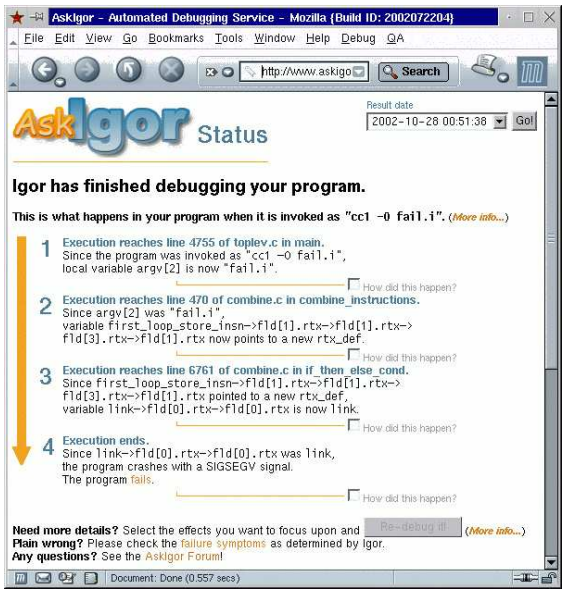

Figure 20.3: The *AskIgor* debugging server

The entire diagnosis is obtained by experimentation alone—no source code is required, and no abstraction from source code takes place.

20.4 Perspectives

Automated program analysis is much more than just analyzing code. In this paper, we show that automated experimentation opens several new perspectives for program analysis: Armed with just an automated test, one can automatically narrow down the causes of specific effects. All this is enabled by the wealth of computing power given to us; yet, we have only begun to combine the different reasoning techniques. This is a great time for investigation and cooperation.

More information about *AskIgor* and related work can be found on our web site

`http://www.st.cs.uni-sb.de/dd/`

Bibliography

[1] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proc. ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 1–10, Charleston, South Carolina, November 2002.