Isolating Cause-Effect Chains with AskIgor

– Tool Demo –

Andreas Zeller
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken, Germany
zeller@acm.org

Abstract

AskIgor is a web service for automatically determining causes of program failures. You submit an executable and two invocations: one where the program fails, and one where it passes. AskIgor compares these two runs and conducts a number of systematic tests to narrow down the set of failure-inducing variable values automatically. This reveals the cause-effect chain of the failure—that is, the variables and values that caused the failure.

In a case study, AskIgor successfully isolated the cause-effect chain for a failure of the GNU C compiler: “Initially, the C program to be compiled 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 the compiler to crash.”

1. About AskIgor

AskIgor (“Ask Igor”) [1] is an experimental automated debugging server that tells you why your program fails. As a user, this is how you work with AskIgor:

Submit a Program. You have a program that shows some repeatable, observable, non-intended behaviour—for instance, the GNU compiler (GCC) crashing on a specific input:

```bash
$ gcc -O fail.c
gcc: Internal compiler error:
    program cc1 got fatal signal 11
$ _
```

You call up the AskIgor Web site (Figure 1) and submit the `cc1` executable—the program that crashes. You also specify two invocations: one where the program fails (as above), and one where it passes; finally, you submit the two required input files.

After submission, you wait for a few minutes—as soon as AskIgor is ready, it notifies you via an e-mail.

Read the Diagnosis. AskIgor presents the diagnosis on its Web page (Figure 2). 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?”).

As stated above, AskIgor is an experimental server; in fact, its main aim is not to provide free debugging services, but to collect a number of buggy programs in order to validate and improve its approach. Nonetheless, AskIgor has already helped to diagnose cause-effect chains for a number of significant programs, the GCC example listed here being the largest so far. Users can judge the quality of AskIgor’s diagnoses; in this demo, I present the latest figures.

2. How Does it Work?

AskIgor does not use program analysis to isolate failure-inducing variables; in fact, it does not analyze the source or machine code at all. Instead, it relies on two principles:

Causes as Differences. A cause is always a difference between a world where the effect occurs and a world where the effect does not occur. Hence, one can narrow down causes from the differences between these two worlds. In our case, these two worlds are the two differing program runs; the cause-effect chain is composed of differences in program state as they propagate from the input to the final outcome. AskIgor instruments an ordinary command-line debugger (GDB) to extract and compare the program states. In the GCC example, such a program state is composed of 78,000 variable values, organized as a memory graph.

Narrowing through Experimentation. The difference between actual program states is still too large to be useful; in the GCC example, about 800 variables differ between the two runs. Therefore, AskIgor narrows down that difference systematically—by creating an intermediate state which takes half of the differences into account. AskIgor injects this intermediate state into the program run and experiments how the program then behaves. Every time the program still fails, or still passes, the difference is reduced by 50%; eventually, only a minimal failure-inducing difference remains—the variable(s) whose value(s) causes the failure to occur.

AskIgor is the first program comprehension tool to exploit these principles; full details of the approach, applied on the GCC case study, are available [2]. I expect that differences and experimentation will have many further benefits in program comprehension and I welcome demo participants to bring in their own ideas and join in the discussion.

3. The Demo

As AskIgor was designed to present its diagnoses with a minimum of manual interaction, demonstrating AskIgor is a no-brainer: submit it, read the diagnosis, fix it. I start with the user’s view—i.e. the program submission; during computation, I explain how the tool works, concluding with the presentation of the actual diagnosis.

Since the above GCC diagnosis takes about 90 minutes to compute, I use a smaller (and faster) example. Participants are invited to bring along buggy programs (as C/C++ source or as Linux executables; see [1] for details) and have AskIgor determine the failure causes on stage.

References