When Abstraction Fails

Andreas Zeller

Saarland University, Saarbrücken, Germany
zeller@cs.uni-sb.de

Abstract. Reasoning about programs is mostly deduction: the reasoning from the abstract model to the concrete run. Deduction is useful because it allows us to predict properties of future runs—up to the point that a program will never fail its specification. However, even such a 100% correct program may still show a problem: the specification itself may be problematic, or deduction required us to abstract away some relevant property. To handle such problems, deduction is not the right answer—especially in a world where programs reach a complexity that makes them indistinguishable from natural phenomena. Instead, we should enrich our portfolio by methods proven in natural sciences, such as observation, induction, and in particular experimentation. In my talk, I will show how systematic experimentation automatically reveals the causes of program failures—in the input, in the program state, or in the program code.

1 Introduction

I do research on how to debug programs. It is not that I am particularly fond of bugs, or debugging. In fact, I hate bugs, and I have spent far too much time on chasing and eradicating them. People might say: So, why don't you spend your research time on improving your specification, model checker, software process, architecture, or whatever the latest and greatest advance in science is. I answer: All of these help *preventing* errors, which is fine. But none can prevent surprises. And I postulate that surprises are unavoidable, that we have to teach people how to deal with them and to set things straight after the fact.

As one of my favorite examples, consider the *sample* program in Fig. 1 on the following page. Ideally, the `sample` program sorts its arguments numerically and prints the sorted list, as in this run ($r_{✓}$):

```
sample 9 8 7  ⇒  7 8 9
```

With certain arguments, `sample` fails (run $r_{✗}$):

```
sample 11 14  ⇒  0 11
```

Surprise! While the output of `sample` is still properly sorted, the output is not a permutation of the input—somehow, a zero value has sneaked in. What is the defect that causes this failure?

In principle, debugging a program like `sample` is easy. Initially, some programmer has created a *defect*—an error in the code. When executed, this defect causes an *infection*—an error in the program state. (Other people call this a *fault*, but I prefer the term

```
/* sample.c -- Sample C program to be debugged */

#include <stdio.h>
#include <stdlib.h>

static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

int main(int argc, char *argv[])
{
    int  i = 0;
    int *a = NULL;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

Fig. 1. The *sample* program (almost) sorts its arguments.

infection, because the error propagates across later states, just like an infection.) When the infection finally reaches a point where it can be observed, it becomes a *failure*—in our case, the zero in the output. Given that a failure has already occurred, it is the duty of the programmer to trace back this cause-effect chain of infections back to the defect where it originated—the defect that caused the failure.

As an experienced programmer, you may be able to walk your way through the source code in Fig. 1 and spot the defect. When it comes to doing so in a general, systematic, maybe even automated way, we quickly run into trouble, though. The difficulty begins with the terms. What do we actually mean when we say “the defect that caused the failure”? What are we actually searching for?

2 Errors are easy to detect, but generally impossible to locate

An *error* is a deviation from what is correct, right, or true. To tell that something is erroneous thus requires a specification of what is correct, right, or true. This can be applied to output, input, state, and code:

Errors in the output. An externally visible error in the program behavior is called a *failure*. Our investigation starts when we determine (or decide) that this is the case.

Errors in the input. For the program *input*, we typically know what is valid and what not, and therefore we can determine whether an input is erroneous or not. If the program shows a failure, and if the input was correct, we know the program as a whole is incorrect.

Errors in the program state. It is between input and output that things start to get difficult. When it comes to the program *state*, we frequently have specifications that allow us to catch infections—for instance, when a pre- or postcondition is violated. Types can be seen as specifications that detect and prevent illegal variable values. Common programming errors, such as buffer overflows or null pointer dereferences, can be specified and detected at compile time.

Errors in the code. Unfortunately, specifications apply only to *parts* of the program state: conditions apply to selected moments in time; types allow a wide range of values; tools can only check for common errors. Therefore, there will always be parts of the state for which correctness is not specified. But if we do not know whether a variable value is correct, we cannot tell whether the code that generated this value is correct. Therefore, we cannot exactly track down the moment the value got infected, and therefore, we cannot locate the defect that caused the failure.

Of course, we can catch errors by simply specifying more. A specification that covers each and every aspect of a program state would detect every single error. Unfortunately, such a specification would ne no less complex and error-prone than the program itself.

In practice, it is the programmer who *decides* what is right upon examining the program—and fixes the program according to this *implied* specification. In such a cases, deciding which part of a program is in error can only be told after the decision has been made and the error has been fixed. Once we know the correct, right, and true code, we can thus tell the defect as a deviation from the corrected code. In other words, *locating a defect is equivalent to writing a correct program.* And we know how hard this is.

3 Causes need not be errors, but can easily be located

While it may be hard to pinpoint an error, the concept of *causality* is far less ambiguous. In general, a *cause* is an event that precedes another event (the *effect*), such that the effect would not have occurred without the cause. For programs, this means that any aspect of an execution causes a failure if it can be altered such that the failure no longer occurs. This applies to input, state, and code:

Causes in the input. We can change the input of the `sample` program from `11 14` (run $r_{✘}$) to `7 8 9` (run $r_{✔}$), and the failure no longer occurs. Hence, we know that the input causes the failure.

One may argue that in any program, the input determines the behavior and thus eventually causes any failure. However, it may be only parts of the input that are relevant. For instance, if we run `sample` with `11`, we find that it is the additional `14` argument which causes the failure:

`sample 11` $\Rightarrow$ `11`

Causes in program state. If we can change some variable during execution such that the failure no longer occurs, we know that the variable caused the failure.

Again, consider the failing `sample` run $r_{\text{✘}}$. We could use an interactive debugger and stop the program at `main()` (Line 28), change `argc` from 2 to 1, and resume execution. We would find an output of `11`, and thus find out that the value of `argc` caused the failure.

As we can see from this example, a cause does not imply an error: The value of `argc` probably is correct with respect to some implied specification; yet, it is tied to the failure.

Causes in the code. All variable values are created by some statement in the code; and thus, there are statements which cause values which again cause failures.

In the `sample` program, there is a statement which exactly does that, and which can (and should) be changed to make the failure no longer occur. The interesting aspect is that we can find that statement from the causes in the program state. If we can find a failure cause in the program state, we can trace it back to the statement which generated it.

Once again, it is important to note that causes and errors are two orthogonal concepts. We can tell an error without knowing whether it is a cause for the failure at hand, and we can tell a cause without knowing whether it is an error. In the absence of a detailed specification, though, we must rely on causality to narrow down those statements which caused the error—in the hope that the defect is among them.

4 Isolating failure causes with automatic experiments

Verifying that something is a cause cannot be done by deduction. We need at least two *experiments:* One with the cause, and one without; if the effect occurs only with the cause, we're set. This implies that we need two runs of the program—one where the failure occurs, and one where the failure does not occur. In debugging, this second run comes at the very end after fixing the defect—if the failure no longer occurs, this verifies that the defect actually caused the original failure.

However, having a passing run $r_{\text{✔}}$ and a failing run $r_{\text{✘}}$ initially is the key for finding causes. The initial difference in the program input causes differences in the program state, which propagate until we see the final difference in the program outcome. By comparing $r_{\text{✔}}$ and $r_{\text{✘}}$, we can extract these differences, and compare them to get a first idea of what caused the failure.

Again, consider the `sample` program. Table 1 on the next page lists the `sample` program states, as well as the differences, as obtained from both $r_{\text{✔}}$ and $r_{\text{✘}}$ when Line 9 was reached. (a and i occur in `shell_sort()` and in `main()`; the `shell_sort()` instances are denoted as a' and i'.)

Formally, this set of 12 differences is a failure cause: If we change the state of $r_{\text{✔}}$ to the state in $r_{\text{✘}}$, we obtain the original failure. However, of all differences, only some may be *relevant* for the failure—that is, it may suffice to change only a *subset* of the variables to make the failure occur. For a precise diagnosis, we are interested in obtaining a subset of relevant variables that is as small as possible.

Variable	Value	
	in $r_{✓}$	in $r_{✗}$
argc	**4**	**5**
argv[0]	`"./sample"`	`"./sample"`
argv[1]	**`"9"`**	**`"11"`**
argv[2]	**`"8"`**	**`"14"`**
argv[3]	**`"7"`**	**`0x0`** (NULL)
i'	1073834752	1073834752
j	1074077312	1074077312
h	1961	1961
size	**4**	**3**

Variable	Value	
	in $r_{✓}$	in $r_{✗}$
i	**3**	**2**
a[0]	**9**	**11**
a[1]	**8**	**14**
a[2]	**7**	**0**
a[3]	1961	1961
a'[0]	**9**	**11**
a'[1]	**8**	**14**
a'[2]	**7**	**0**
a'[3]	1961	1961

Table 1. One of the state differences between $r_{✓}$ and $r_{✗}$ causes *sample* to fail.

Delta Debugging [3] is a general procedure to obtain such a small subset. Given a set of differences (such as the differences between the program states in Fig. 1), Delta Debugging determines a *relevant subset* in which each remaining difference is relevant for the failure to occur. To do so, Delta Debugging systematically and automatically *tests* subsets and narrows down the difference depending on the test outcome, as sketched in Fig. 2. Overall, Delta Debugging behaves very much like a binary search.

Originally, Delta Debugging was designed for program inputs. However, one may consider a program state as an input to the remainder of the program execution; hence, it is pretty straight-forward to apply Delta Debugging on program states to isolate causes. Applied on the differences in Table 1, Delta Debugging would result in a first test that

- runs $r_{✓}$ up to Line 9,
- applies *half* of the differences on $r_{✓}$—that is, it sets `argc`, `argv[1]`, `argv[2]`, `argv[3]`, `size`, and `i` to the values from $r_{✗}$—, and
- resumes execution and determines the outcome.

This test results in the same output as the original run; that is, the six differences applied were not relevant for the failure. With this experiment, Delta Debugging has narrowed down the failure-inducing difference to the remaining six differences. Repeating the search on this subset eventually reveals one single variable, `a[2]`, whose zero value is failure-inducing: If, in $r_{✓}$, we set `a[2]` from 7 to 0, the output is `0 8 9`—the failure occurs. We can thus conclude that the zero being printed is caused by `a[2]`—which we can confirm further by changing `a[2]` in $r_{✗}$ from 0 to 7, and obtaining the output `7 11`. Thus, in Line 9, `a[2]` being zero causes the `sample` failure.

The idea of determining causes by experimenting with mixed program states (rather than by analyzing the program or its run, for instance) may seem strange at first. Yet, the technique has been shown to produce useful diagnoses for programs as large as the GNU compiler (GCC). As detailed in [2], scaling up the general idea, as sketched here, requires capturing and comparing program states as *memory graphs* [4]. Also, Delta Debugging must do more than simple binary search; it needs to cope with interferences of multiple failure-inducing elements as well as with unresolved test outcomes [3].

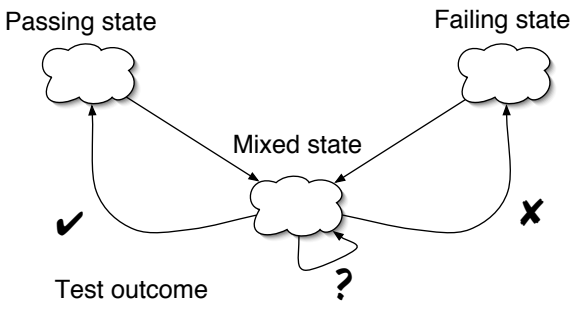

Fig. 2. Narrowing down state differences. By assessing whether a mixed state results in a passing (✔), a failing (✘), or an unresolved (**?**) outcome, Delta Debugging isolates a relevant difference.

5 Locating the statements that cause the failure—automatically

The tricky question is now: How do we get from failure-causing states to failure-causing statements? One straight-forward way might be to look at the statements which created the value. Alas, we won't find such a statement for `a[2]`; it is never assigned a value before Line 9.

However, it turns out that at the start of `main()`, in Line 28, it is not `a[2]` which causes the failure, but *argc*—if we change the value of `argc` from 4 (its value in $r_{✘}$) to 3 (its value in $r_{✔}$), the failure no longer occurs. Since initially, `argc` caused the failure, and later, `a[2]`, there must have been some moment in time where this transition from `argc` to `a[2]` took place. This transition can be isolated using binary search over time: it takes place at Line 35, at the call

```
shell_sort(a, argc);
```

This is where `argc` stops to be a cause, and `a[2]` begins. This transition implies that Line 35 causes `a[2]` to cause the failure—or, in other words, that we can change Line 35 to make the failure no longer occur. Line 35 is a failure cause.

So, let us focus on Line 35 to see whether it is not only a cause, but in fact, erroneous. Let us assume that in the declaration `shell_sort(int a[], int size)`, the parameter `size` stands for the number of elements in `a[]`. Then, the call in Line 35 is wrong—simply because `argc` is not the number of elements in `a[]`, but off by one. The correct call would be

```
shell_sort(a, argc - 1);
```

By changing the statement, we can re-run the test to see whether the failure still occurs. It does not; hence, we have proven that the defect actually caused the failure—and successfully fixed the program.

In this example, the cause transition from `argc` to `a[2]` occurred right at the place of the defect. As a programmer, though, I may also have decided to change `shell_sort()` instead such that `size` is the number of elements in `a` plus one. I

Fig. 3. ASKIGOR with a diagnosis for *sample*

could also decrease the value of `argc` or introduce a new variable `arguments` initialized with `argc - 1`. This number of alternatives shows that it is difficult to *predict* an exact change, say, in an evaluation. Therefore, when evaluating whether cause transitions are effective in locating defects, one uses a measure of *closeness:* If we cannot predict the exact location of the defect, how close are we in locating it?

To evaluate a defect locator, one thus ranks the statements of the program according to their likelihood to be defective. In our case, we'd rank the locations of cause transitions at the top, followed by "close" locations—that is, those related by one control or data dependency—and followed by less close locations by doing an exhaustive breadth-first search along the system dependency graph. The assumption is that a programmer starts with the most likely locations (at the top) and then walks down the list until he or she finds the defect. In a case study [1], it turned out that cause transitions are the best defect locators available—they locate the failure-inducing defect twice as well as the best methods known so far. The technique is implemented as part of the ASKIGOR debugging server (Fig. 3).

Yet, we have just begun to explore the idea of making experimenting a part of program analysis. There is still a long way to go before these techniques can become part of the mainstream: in particular, extracting and mixing program states becomes a challenge if the program is deeply interwoven with its environment. On the other hand, having automated diagnoses is not only convenient for the programmer, but also may enable new generations of self-aware systems: Think of a Web server, for instance, that automatically determines a failure cause in its own code, and thus disables the appropriate configuration module—at least as a temporary fix until the code is corrected.

6 Conclusion: Prevent errors *and* prepare for surprises

Why focus on cure, when prevention is so much better? Of course, we should continue to strive for systems that have as few defects as possible. But this must not mean to neglect the cure altogether. In a world where software systems become more and more complex, we must be prepared for surprises. And a surprise is exactly what happens when the given abstraction fails, or where there simply is no abstraction that could tell what's right and what's wrong.

Program analysis has long been based on abstraction alone—deducing predictions from the program code that hold for future program runs. To analyze *past* program runs, though, requires a much wider portfolio of techniques—simply because there is much more data to take into account: Besides program code, we can look at actual runs, test outcomes, version histories—any artifact created during development is welcome. And if *induction* to derive common patterns from all these instances is not enough, we can use *experimentation* to generate even more. Fortunately for us, we now have the computational power available to apply all these techniques. What we need is a confluence of static and dynamic analysis, of deduction and induction techniques—to foster the understanding of today's programs, and to bring surprises and their damage to a minimum.

Acknowledgments. Thanks to all who gave me feedback on earlier instances of this talk. Christian Lindig and Stephan Neuhaus gave valuable comments on this paper.

Read more

[1] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proc. International Conference on Software Engineering (ICSE)*, St. Louis, Missouri, May 2005.

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

[3] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[4] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In Stephan Diehl, editor, *Proc. of the International Dagstuhl Seminar on Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 191–204, Dagstuhl, Germany, May 2002. Springer-Verlag.

All papers and project news are available online at

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