CHAPTER

12 Causes and Effects

Deduction, observation, and induction are all good in finding *potential* defects. However, none of these techniques alone is sufficient to determine a *failure cause*. How does one identify a cause? How does one isolate not only *a* cause but *the* actual cause of a failure? This chapter lays the groundwork on how to find failure causes systematically—and automatically.

12.1 CAUSES AND ALTERNATE WORLDS

Anomalies and defects, as discussed in the previous chapters, are all good starting points in a debugging session. However, we do not know yet whether these actually *cause* the failure in question.

If we say "a defect causes a failure," what does "cause" mean? Generally speaking, a *cause* is an event preceding another event without which the event in question (the *effect*) would not have occurred. Thus, a defect causes the failure if the failure would not have occurred without the defect.

Because most of debugging is the search for a defect that causes the failure, we must understand how to search for cause–effect relationships. That is, to search for *causality*—with the idea that once we found a cause, the defect is not far away.

In natural and social sciences, causality is often difficult to establish. Just think about common disputes such as "Did usage of the butterfly ballot in West Palm Beach cause George W. Bush to be president of the United States?"; "Did drugs cause the death of Elvis Presley?"; "Does human production of carbon dioxide cause global warming?"

To determine whether these are actually causes, formally we would have to repeat history *without the cause in question*—in an alternate world that is as close as possible to ours except for the cause. Using this *counterfactual* model of causality, a cause becomes a *difference* between the two worlds (Figure 12.1):

- A world where the effect occurs.
- An *alternate* world where the effect does not occur.

We know already what our actual world looks like. However, if in alternate worlds Albert Gore had been president, Elvis were alive, and global warming were less

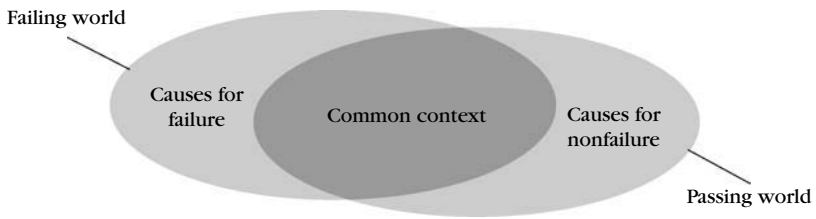

FIGURE 12.1

Causes as differences between alternate worlds.

(and not changing anything else), we would know that butterfly ballots, drugs, and carbon dioxide had been actual causes for the given effects.

Unfortunately, we cannot repeat history like an experiment in a lab. We have to speculate about what *would* have happened. We can have all experts in the world agree with our speculation, but in the real, nondeterministic, and, above all, nonrepeatable world, one can never absolutely know whether a probable cause is a cause. This is why one can always come up with a new theory about the true cause, and this is why some empirical researchers have suggested dropping the concept of causality altogether.

In our domain of computer science, though, things are different. We can easily repeat program runs over and over, change the circumstances of the execution as desired, and observe the effects. Given the right means (Chapter 4), the program execution is under (almost) total control and is (almost) totally deterministic. (The "almost" is there because the execution may still be determined by physical effects, as described in Section 4.3.8 in Chapter 4.)

Scientists frequently use computers to determine causes and effects in models of the real world. However, such causes and effects run the danger of being inappropriate in the concrete, because the model may have abstracted away important aspects. If we are determining causes and effects in the program itself, though, we keep abstraction to a minimum. Typically, we only abstract away the irreproducible physical effects. Minimal abstraction implies minimal risk. Thus, among all scientific disciplines *debugging is the one that can best claim to deal with actual causality.* In the remainder of this chapter, we shall thus address the key question:

> How do I know something causes the failure in question?

12.2 VERIFYING CAUSES

How do we check whether some anomaly—or, more generally, any property of a program run—causes the failure in question? The actual world in which the effect occurs is our real world—with the effect, the failing run, occurring before our eyes.

The *alternate world,* though, is what we need to show that some property causes the failure. To show causality, we must *set up an experiment with an alternate world* in which the property does not occur. If in this alternate world the failure does not occur either, we have shown that the property caused the failure.

As an example, consider a program that emits a warning message (say, "Configuration file not found") and then fails. How do we show that the missing configuration file causes the failure? We set up an alternate world in which a configuration file is present. In other words, we set up an *experiment* to support or refute our hypothesis about causality. If (and only if) in our experiment the failure no longer occurs have we shown causality (i.e., that the missing file caused the failure).

This reasoning may sound trivial at first ("Of course, we need such an experiment!"), but having such an explicit verification step is crucial for avoiding *fallacies.* In our example, it would only be natural to assume that the warning message is somehow connected to the failure—especially if the warning is all we can observe—and thus attempt to resolve the warning in the hope of resolving the failure.

This type of reasoning is called *post hoc ergo propter hoc* ("after this, therefore because of this"). This means that an anomaly has occurred *before* the failure, and therefore the anomaly must have caused the failure. However, it may well be that the warning is totally unrelated to the failure. In that case, resolving the warning will cost us precious time. Therefore, any systematic procedure will first determine causality by an experiment, as described previously.

12.3 CAUSALITY IN PRACTICE

The following is a somewhat more elaborate example. Consider the following piece of C code.

```
a = compute_value();
printf("a = %d\n", a);
```

This piece of code invokes the `compute_value()` function, assigns the result to the variable `a`, and prints the value of `a` on the console. When executed, it prints `a = 0` on the console, although `a` is not supposed to be zero. What is the cause for `a = 0` being printed?

Deducing from the program code (see Chapter 7), we may reason that if variable `a` is zero we must examine the origin of this value. Following back the dependences from `a`, we find that the last assignment was from `compute_value()`, and thus we might investigate how `compute_value()` can return a zero value. Unfortunately, it turns out that `compute_value()` is not supposed to return zero. Thus, we may proceed digging into the `compute_value()` code to find out how the zero came to be.

Unfortunately, reasoning alone does not suffice for proving causality. We must show by experiment that if a cause is not present, an effect is not present. Therefore, we must show that `a` being zero is the cause for `a = 0` being printed. Later on, we

would show that `compute_value()` returning zero is the cause for `a` being zero—and each of these causalities must be shown by an experiment (or, at least, additional observation).

At this point, this may seem like nitpicking. Isn't it *obvious* that `a` is zero? After all, we print its value on the console. Unfortunately, "obvious" is not enough. "Obviously," the program should work, but it does not. Thus, we can trust nothing, and especially not the obvious.

Let's then attempt to show causality using scientific method. We set up a hypothesis:

a *being zero is the cause for* a = 0 *being printed*.

To show that this hypothesis holds, we must set up an experiment in which `a` is not zero, and in which `a = 0` is not being printed. Let's do so by inserting a little assignment `a = 1` into the code. This results in our first *alternate world*:

```
a = compute_value();
a = 1;
printf("a = %d\n", a);
```

If the program now prints `a = 1`, we know that `a` being zero was the cause for `a = 0` being printed. However, if we execute this piece of code we find that `a = 0` is still being printed, regardless of the inserted assignment.

This is weird. How can this happen? We set up a new hypothesis:

a = 0 *is being printed regardless of the value of* a.

To prove this hypothesis, we could set `a` to various values other than `1`—and we would find that the hypothesis always holds. This means that there must be something wrong with the `printf()` invocation. And there is. `a` is declared a *floating-point* variable:

```
   double a;
⋮
   a = compute_value();
   a = 1;
   printf("a = %d\n", a);
```

However, the argument `"%d"` makes `printf()` expect an *integer* value as a next argument. If we pass a floating-point value instead, this means that only the first four bytes of the floating-point representation are being read—and interpreted as an integer. (In fact, this type of mistake is so common that many compilers issue a warning when the `printf()` format string does not match the later arguments.) What happens on this specific machine is that the first four bytes of the internal representation of `1.0` (and any other small integer) are all zero—and thus `a = 0` is printed regardless of the value of `a`.

But all this, again, is yet only reasoning. Our working hypothesis becomes:

The format `"%d"` *is the cause for* `a = 0` *being printed*.

To prove the hypothesis, we must again set up an experiment in which the cause does not occur. In other words, we must alter the program to make the failure go away. A proper format for floating-point values in `printf()` is `"%f"`. Let's alter the format to this value:

```
a = compute_value();
printf("a = %f\n", a);
```

Now that the cause `"%d"` is no longer present, the actual value of `a` is being printed on the console. That is, the effect of printing `a = 0` is gone. This means that `"%d"` actually was the cause of `a = 0` being printed. In other words, `"%d"` was the defect that caused the failure. Our final hypothesis has become a *theory* about the failure cause. Note how the use of scientific method (Chapter 6) prevents fallacies from the start, as every hypothesis (about a failure cause) must be verified by an experiment (with an alternate world in which the cause does not occur).

As pointed out in Section 12.1, detecting causes in the real world is difficult, essentially because one cannot turn back history and see what would have happened in the alternate world. This is why the counterfactual definition of causality is often deemed too restrictive. In the context of debugging, though, we can repeat runs over and over. In fact, conducting experiments with alternate worlds is a necessary effect of applying scientific method. Thus, in debugging, experiments are the only way to show causality. Deduction and speculation do not suffice.

12.4 FINDING ACTUAL CAUSES

Now that we have discussed how to *verify* a cause, let's turn to the central problem: How do we find a failure cause? It turns out that finding *a* cause is trivial. The problem is to find *the* cause among a number of alternatives.

In debugging, as in experimental science, the only way to determine whether something is a cause is through an experiment. Thus, for example, only by *changing* the program in Section 6.3 in Chapter 6 could we prove that the defect was actually the *cause* of the failure.

This conjunction of causes and changes raises an important problem. Just as there are infinitely many ways of writing a program, there are infinitely many ways of changing a program such that a failure no longer occurs. Because each of these changes implies a failure cause, there are *infinitely many failure causes*. For example, how can we say that something is *the* defect or *the* cause of a failure, as `"%d"` in Section 12.3?

- We could also say that the `printf()` statement as a whole is a cause for printing `a = 0`, because if we remove it nothing is printed.
- Anomalies, as discussed in Chapter 11, are a cause of a failure because without anomalies (i.e., in a "normal" run) the failure does not occur.
- We can treat the entire program code as a cause, because we can rewrite it from scratch such that it works.

- Electricity, mathematics, and the existence of computers are all failure causes, because without them, there would be no program run and thus no failure.

This multitude of causes is unfortunate and confusing. In debugging, and especially in automated debugging, we would like to point out a single failure cause, not a multitude of *trivial* alternatives.

To discriminate among these alternatives, the concept of the *closest possible world* comes in handy. A world is said to be "closer" to the actual world than another if it resembles the actual world more than the other does. The idea is that now *the* cause should be a *minimal difference* between the actual world where the effect occurs and the alternate world where it would not (Figure 12.1). In other words, the alternate world should be *as close as possible*. Therefore, we define an *actual cause* as a difference between the actual world where the effect occurs, and the closest possible world where it would not.

Another way of thinking about an actual cause is that *whenever we have the choice between two causes we can pick the one of which the alternate world is closer.* Consequently, `"%d"` is *the* defect, but the `printf()` statement is not—because altering just the format string is a smaller difference than removing the `printf()` statement. Likewise, the absence of electricity would result in a world that is quite different from ours. Thus, electricity would not qualify as an actual failure cause. This principle of picking the closer alternate world is also known as *Ockham's Razor*, which states that whenever you have competing theories for how some effect comes to be, pick the simplest.

12.5 NARROWING DOWN CAUSES

Let's now put these general concepts of causality into practice. Given some failure (i.e., the effect), how do we find an actual cause? A simple strategy works as follows:

1. Find an alternate world in which the effect does not occur.
2. Narrow down the initial difference to an actual cause, using scientific method (Chapter 6).

If you think that this sounds almost too trivial, you are right. The alternate world is where the effect does not occur—is not this just what we aim at? Think about a defect causing the failure, for instance. If we have an alternate world in which the defect does not occur, we are already set. Why bother dealing with the differences to the real world if we already know what the alternate world is supposed to be?

The trick is that the alternate world need not be a world in which the program has been corrected. It suffices that the failure does not occur—which implies that there is some other *difference* either in the program input or its execution that eventually causes the differing behavior with respect to the failure. The challenge is to *identify* this initial difference, which can then be narrowed down to an actual cause.

12.6 A NARROWING EXAMPLE

The following is a little example that illustrates this approach. When I give a presentation, I use a little shareware program on my laptop such that I can remote-control the presentation with my Bluetooth phone. (Sometimes it's fun to be a nerd.) Having upgraded this program recently, I found that it quit after moving to the next slide. I exchanged a few emails with the author of the shareware. He was extremely helpful and committed, but was not able to reproduce my problem. In his setting (and in the setting of the other users), everything worked fine.

To narrow down the cause, I searched for an alternate world in which the failure did not occur. It turned out that if I created a new user account from scratch, using all of the default settings, the program worked fine. Thus, I had a workaround—but I also had a *cause*, as this alternate account (the alternate world) differed from my account in a number of settings and preferences.

Just having a cause, though, did not help me in fixing the problem. I wanted an *actual* cause. Thus, I had to narrow down the difference between the accounts, and so I copied setting after setting from my account to the new account, checking each time whether the failure would occur. In terms of scientific method, the hypothesis in each step was that the respective setting caused the problem, and with each new copied setting not showing the failure I disproved one hypothesis after another.

Finally, though, I was successful: copying the *keyboard settings* from my account to the new account caused the failure to occur. Mostly living in Germany, I occasionally have to write text in German, using words with funny characters such as those in "Schöne Grüße" (best regards). To type these characters quickly, I have crafted my own keyboard layout such that simple key combinations such as Alt+O or Alt+S give me the ö or ß character, respectively. Copying this layout setting to the new user account resulted in the failure of the shareware program.

Thus, I had an actual cause—the difference between the previous setting and the new setting—and this diagnosis was what I emailed the shareware author (who had explored some other alternatives in the meantime). He committed to support such hand-crafted layouts in the future, and everybody was happy.

12.7 THE COMMON CONTEXT

As the example in Section 12.6 illustrates, we do not necessarily need an alternate world in which the defect is *fixed*. It suffices to have some alternate world in which the failure *does not occur*—as long as we can narrow down the *initial difference* to an actual cause.

This initial difference sets the frame in which to search for the actual cause. Aspects that do not differ will be part of the *common context* and thus never changed nor isolated as causes. This context is much larger than may be expected.

Our common context includes, for instance, the fact that the program is executed, and all other facts required to make the execution possible. One can also think about the common context as defining *necessary conditions* that must be satisfied by every alternate world. Anything that is not part of the common context can *differ*, and thus sets the *search space*—such as the settings in my user account.

In many cases, we have the choice between multiple alternate worlds. For instance, I may find out that some earlier version of the shareware program works nicely, such that I could narrow down the difference between the old and new version—with the settings unchanged. I could try different devices and leave everything else unchanged. Perhaps the failure occurs only with specific devices? The choice of an alternate world sets the search space in which to find causes. Whatever alternate world is chosen, one should strive to keep it *as similar as possible to the actual world*—simply because Ockham's Razor tells you that this gives you the best chance of finding the failure cause.

12.8 CAUSES IN DEBUGGING

The concepts *actual cause* and *closest possible world* are applicable to all causes—including the causes required for debugging. Thus, if we want to find the actual cause for a program failure we have to search for the *closest possible world in which the failure does not occur*.

- *Input.* The actual failure cause in a *program input* is a minimal difference between the actual input (where the failure occurs) and the closest possible input where the failure does not occur.
- *State.* The actual failure cause in a *program state* is a minimal difference between the actual program state and the closest possible state where the failure does not occur.
- *Code.* The actual failure cause in a *program code* is a minimal difference between the actual code and the closest possible code where the failure does not occur.

All of these failure causes must be *verified* by two experiments: one in which effect and failure occur and one in which they do not. Once a cause has been verified, valuable information for debugging is available.

- *Causes are directly related to the failure.* As a failure is an effect of the cause, it only occurs when the cause occurs. Neither defects, (Chapter 7), nor anomalies, (Chapter 11) are as strongly related to the failure.
- *Failure causes suggest fixes.* By removing the failure cause, we can make the failure disappear. This may not necessarily be a *correction*, but it is at least a good *workaround.*

Both properties make causes excellent starting points during debugging, which is why in the remainder of this book we will explore how to isolate them automatically.

12.9 CONCEPTS

Of all circumstances we can observe during debugging, *causes* are the most valuable.

A cause is an event preceding another event without which the event in question (the *effect*) would not have occurred.

A cause can be seen as a *difference* between two worlds—a world in which the effect occurs and an *alternate* world in which the effect does not occur.

How To

To show causality, set up an experiment in which the cause does not occur. Causality is shown if (and only if) the effect does not occur either.

To find a cause, use a scientific method to set up hypotheses on possible causes. Verify causality using experiments.

An *actual cause* is the difference between the actual world and the *closest possible world* in which the effect does not occur.

The principle of picking the closest possible world is also known as *Ockham's Razor*, which states that whenever you have competing theories for how some effect comes to be, pick the simplest.

To find an actual cause, narrow down an initial difference via scientific method.

A *common context* between actual worlds excludes causes from the search space.

12.10 FURTHER READING

The definitions of cause and effect in this book are based on *counterfactuals*, because they rely on assumptions about nonfacts. The first counterfactual definition of causes and effects is attributed to Hume (1748): "If the first object [the cause] had not been, the second [the effect] never had existed." The best-known counterfactual theory of causation was elaborated by Lewis (1973), refined in 1986.

Causality is a vividly discussed philosophical field. In addition to the counterfactual definitions, the most important alternatives are definitions based on *regularity* and *probabilism*. I recommend Zalta (2002) for a survey and Pearl (2000) for an in-depth treatment.

Ockham's Razor is the principle proposed by William of Ockham in the 14th century: "Pluralitas non est ponenda sine neccesitate," which translates as "plurality shouldn't be posited without necessity." A modern interpretation is: "If two theories explain the facts equally well, the simpler theory is to be preferred." Or just: "Keep it simple." The principle was stated much earlier by Aristotle: "For if the consequences are the same it is always better to assume the more limited antecedent."

According to Bloch (1980), *Hanlon's Razor* "Never attribute to malice that which is adequately explained by stupidity" was coined by the late Robert J. Hanlon of Scranton, Pensylvania. (This phrase or very similar statements have been

attributed to William James, Napoleon Bonaparte, Richard Feynman, Johann Wolfgang von Goethe, Robert Heinlein, and others.) Reportedly, Hanlon was a winner in a contest to come up with further statements similar to *Murphy's Law*: "If it can go wrong, it will."

EXERCISES

12.1 Suppose you wish to find out whether:

- Elvis died of an overdose of drugs.
- The butterfly ballot cost Al Gore the White House.
- Global warming is caused by carbon dioxide.

Which experiments would you need to support your views?

12.2 Consider the experiment in Section 6.3 in Chapter 6. In each step:

(a) What is the hypothesis about the failure cause?
(b) How does the hypothesis verify causality?

12.3 Consider the failure of the `bigbang` program in Example 8.3.

(a) List three *actual* failure causes.
(b) List three failure causes that are not actual causes.
(c) Where would you correct the program? Why?

12.4 Be creative. Write a failing program with:

(a) A *failure cause* that looks like an *error* (but is not).
(b) An *error* that looks as if it *caused the failure* (but does not).

12.5 Site *A* and site *B* each send a virus-infected email to site *C*. *A*'s email arrives first, infecting *C*. Using the counterfactual definition, what is the cause:

(a) For site *C* being infected?
(b) For site *C* being infected right after *A*'s email arrives?

Try to find your own answer first, and then look at the discussion (and further tricky examples) in Zalta (2002).

12.6 What are the relationships among failing world, passing world, initial difference, and cause?

12.7 Explain the meaning of *closest possible world in which a failure does not occur.* What type of failure causes do we distinguish, and how do we verify them? Illustrate using examples.

12.8 Each of the following statements is either true or false.

- If *c* is a cause, and *e* is its effect, then *c* must precede *e*.
- If *c* is a circumstance that causes a failure, it is possible to alter *c* such that the failure no longer occurs.

- If some cause *c* is an actual cause, altering *c* induces the smallest possible difference in the effect.
- Every failure cause implies a possible fix.
- For every failure there is exactly one actual cause.
- A failure cause can be determined without executing the program.
- A failure is the difference to the closest possible world in which the cause does not occur.
- If I observe two runs (one passing, one failing) with a minimal difference in input, I have found an actual failure cause.
- A minimal and successful correction proves that the altered code was the actual failure cause.
- Increasing the common context between the possible worlds results in smaller causes.

12.9 "Given enough evidence, an anomaly can qualify as a cause." Discuss.

> When you have eliminated the impossible, whatever remains, however improbable, must be the truth.
>
> –Sherlock Holmes, in A. Conan Doyle
> *The Sign of Four* (1890)