CHAPTER

13 Isolating Failure Causes

This is the chapter that automates most debugging. We show how delta debugging isolates failure causes automatically—in the program input, in the program's thread schedule, and in the program code. In the best case, the reported causes immediately pinpoint the defect.

13.1 ISOLATING CAUSES AUTOMATICALLY

Narrowing down causes as described in Chapter 12 can be tedious and boring—when conducted manually, that is. Therefore, we should aim at narrowing down failure causes *automatically.*

In principle, narrowing down a cause can be easily automated. All it takes is

- an *automated test* that checks whether the failure is still present,
- a means of *narrowing down the difference*, and
- a *strategy* for proceeding.

With these ingredients, we can easily *automate the scientific method* involved. We have some automaton apply one difference at a time; after each difference, the automaton tests whether the failure now occurs. Once it occurs, we have narrowed down the actual cause.

Consider the keyboard layout example from Section 12.6 in Chapter 12, in which a specific keyboard layout setting caused a presentation shareware to fail. In this example, automation translates to the following points:

- The automated test starts the presentation shareware and checks for the failure.
- The means of narrowing down the difference is copying settings from one account to another.
- The strategy for proceeding could be to copy one setting at a time.

Proceeding one difference at a time can be very time consuming, though. My keyboard layout, for instance, has definitions for 865 key combinations. Do I really want to run 865 tests just to learn that I should not have Alt+O defined? What we need here is a more effective strategy—and this brings us to our key question:

HOW DO I ISOLATE FAILURE CAUSES AUTOMATICALLY?

13.2 ISOLATING VERSUS SIMPLIFYING

In Chapter 5, we saw how to leverage automated tests to simplify test cases quickly, using delta debugging. One could think of applying this approach toward simplifying the difference between the real world and the alternate world—that is, to find an alternate world the difference of which to the real world is as "simple" or as close as possible. In practice, this means trying to remove all differences that are not relevant for producing the failure—that is, to bring the alternate world as close as possible to the real world. In the remaining difference, each aspect is relevant for producing the failure—that is, we have an actual cause.

When we are thinking about narrowing down differences, though, there is a more efficient approach than simplifying, called *isolating*. In simplifying, we get a test case where each single circumstance is relevant for producing the failure. Isolating, in contrast, produces a *pair* of test cases—one passing the test, one failing the test—with a minimal difference between them that is an actual cause.

Let's highlight isolation using an example. In Example 5.5, we saw how the *ddmin* algorithm eventually simplifies a failure-inducing HTML line from 40 characters down to 8 characters in 48 tests. In the result, a `<SELECT>` tag, every single character is relevant for producing the failure.

Isolating works in a similar fashion, each time a test case fails, the smaller test case is used as the new failing test case. However, we do not just remove circumstances from the failing test case but *add circumstances to the passing test case*, and thus may obtain a new (larger) passing test case. Figure 13.1 highlights the difference between simplification and isolation. Simplifying results in a simplified failing test case, whereas isolation results in a passing and a failing test case with a minimal difference.

Example 13.1 shows how this works on the HTML input of the Mozilla example. Starting with the empty passing input (bottom) and the 80-character failing input (top), we first remove half the characters—as in *ddmin*. The test passes, and this is where isolation shows the difference. We use the half of the input as a new passing test case and thus have narrowed down the difference (the cause) to the second half of the characters. In the next step, we add half of this half to the passing test case, which again passes the test.

Continuing this pattern, we eventually end up in a minimal difference between the original failing input

```
<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>
```

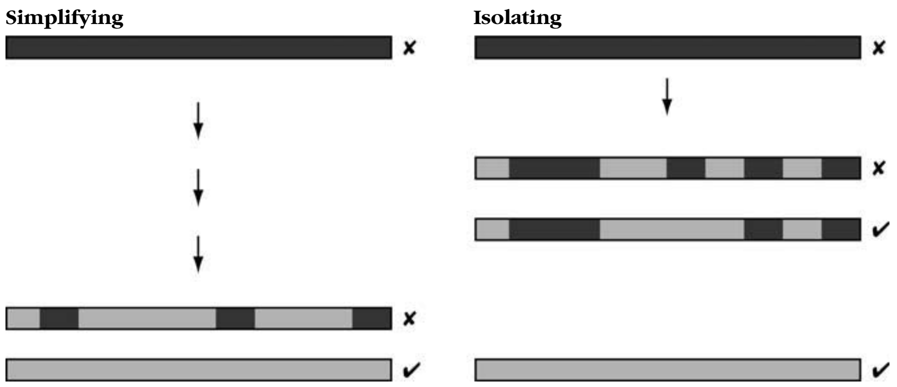


FIGURE 13.1

Simplifying versus isolating. While simplifying, we bring the failing configuration (✗) as close as possible to the (typically empty) passing configuration (✓). When isolating, we determine the smallest difference between the two, moving the passing, as well as the failing, configuration.

EXAMPLE 13.1: Isolating a failure-inducing difference. After six tests, the < is isolated as failure cause

```
Input: <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨40 characters ⟩ ✗
       <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨ 0 characters ⟩ ✓

     1 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨20 characters ⟩ ✓
     2 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨30 characters ⟩ ✓
     3 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨35 characters ⟩ ✓
     4 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨37 characters ⟩ ✓
     5 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨38 characters ⟩ ✓
     6 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨39 characters ⟩ ✓

Result: <
```

and the new passing input

```
SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>
```

The difference is in the first character: adding a < character changes the `SELECT` text to the full HTML `<SELECT>` tag, causing the failure when being printed. This example demonstrates the basic difference between simplification and isolation.

- *Simplification* means to make each part of the simplified test case relevant. Removing any part makes the failure go away.
- *Isolation* means to find one relevant part of the test case. Removing this particular part makes the failure go away.

As an allegory, consider the flight test from Section 5.1 in Chapter 5. Simplifying a flight test returns the set of circumstances required to make the plane fly (and eventually crash). Isolating, in contrast, returns two sets of circumstances that differ by a minimum—one set that makes the plane fly (that is, the "passing" outcome) and a set that makes the plane crash (the "failing" outcome). The difference is a failure cause, and being minimal, it is even an actual failure cause.

In general, isolation is much more efficient than simplification. If we have a large failure-inducing input, isolating the difference will pinpoint a failure cause much faster than minimizing the test case. In Example 13.1, isolating requires only 5 tests, whereas minimizing (Example 5.8) requires 48 tests.

The greater efficiency of isolation comes at a price, though. An isolated difference can come in a *large context*, which may require more effort to understand—especially if the isolated cause is not an error. Reconsidering the flight test example, assume we isolate that switching on the cabin light causes the crash. If the light stays off, the plane lands perfectly. Switching on the cabin light is standard procedure, and thus we still have to find out how this event interacts with the context such that it leads to the crash. With minimization, we simplify the context as a whole. We still find that the cabin light is relevant for the crash, but we only keep those other events that are also relevant (e.g., the short-circuit in the cabin light cable).

13.3 AN ISOLATION ALGORITHM

How do we automate isolation? It turns out that the original *ddmin* algorithm, as discussed in Section 5.5 in Chapter 5, can easily be extended to compute a minimal difference rather than a minimal test case. In addition to reducing the failing test case $c_{✘}$ whenever a test fails, we now *increase* the passing test case $c_{✔}$ whenever a test passes. The following is what we have to do to extend *ddmin.*

1. Extend-*ddmin* -such that it works on two sets at a time:
   - The passing test case $c'_{✔}$ that is to be maximized (initially, $c'_{✔} = c_{✔} = \emptyset$ holds).
   - The failing test case $c'_{✘}$ that is to be minimized (initially, $c'_{✘} = c_{✘}$ holds).

   These two sets are the worlds between which we narrow down the difference.
2. Compute subsets Δ_i as subsets of $\Delta = c'_{✘} \setminus c'_{✔}$ (instead of subsets of $c'_{✘}$).
3. In addition to testing a removal $c'_{✘} \setminus \Delta_i$, test an *addition* $c'_{✔} \cup \Delta_i$.
4. Introduce new rules for passing and failing test cases:
   - *Some removal passes:* If $c'_{✘} \setminus \Delta_i$ passes for any subset Δ_i, then $c'_{✘} \setminus \Delta_i$ is a larger passing test case. Continue reducing the difference between $c'_{✘} \setminus \Delta_i$ and $c'_{✘}$.
   - *Some addition fails:* This is the complement to the previous rule. If $c'_{✔} \cup \Delta_i$ fails for any subset Δ_i, then $c'_{✔} \cup \Delta_i$ is a smaller failing test case. Continue reducing the difference between $c'_{✔}$ and $c'_{✔} \cup \Delta_i$.
   - *Some removal fails:* This rule is the same as in *ddmin*. If $c'_{✘} \setminus \Delta_i$ fails for any subset Δ_i, then $c'_{✘} \setminus \Delta_i$ is a smaller failing test case. Continue reducing the difference between $c'_{✔}$ and $c'_{✘} \setminus \Delta_i$.

- *Some addition passes:* Again, this is the complement to the previous rule. If $c'_{✔} \cup \Delta_i$ passes for any subset Δ_i, then $c'_{✔} \cup \Delta_i$ is a larger passing test case. Continue reducing the difference between $c'_{✔} \cup \Delta_i$ and $c'_{✘}$.
- *Increase granularity:* This rule is as in *ddmin*, but applies only if all tests are unresolved. Increase the granularity and split $c_{✘}$ into 4 (8, 16, and so on) subsets.

The full algorithm, named *dd*, is shown in Example 13.2. It builds on the definitions used for *ddmin* and is called the *general delta debugging algorithm.* The

EXAMPLE 13.2: The *dd* algorithm in a nutshell

Let a program's execution be determined by a set of circumstances called a *configuration*. By $\mathcal{C}$, we denote the set of all *changes* between configurations.
Let *test* $: 2^{\mathcal{C}} \to \{✘, ✔, ?\}$ be a testing function that determines for a configuration $c \subseteq \mathcal{C}$ whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (?).

Now, let $c_{✔}$ and $c_{✘}$ be configurations with $c_{✔} \subseteq c_{✘} \subseteq \mathcal{C}$ such that *test* $(c_{✔}) = ✔ \wedge$ *test* $(c_{✘}) =$ ✘. $c_{✔}$ is the "passing" configuration (typically, $c_{✔} = \emptyset$ holds) and $c_{✘}$ is the "failing" configuration.

The *general delta debugging algorithm dd* $(c_{✔}, c_{✘})$ isolates the failure-inducing difference between $c_{✔}$ and $c_{✘}$. It returns a pair $(c'_{✔}, c'_{✘}) = dd(c_{✔}, c_{✘})$ such that $c_{✔} \subseteq c'_{✔} \subseteq c'_{✘} \subseteq c_{✘}$, *test* $(c'_{✔}) = ✔$, and *test* $(c'_{✘}) = ✘$ hold and $c'_{✘} \setminus c'_{✔}$ is a *relevant difference*—that is, no single circumstance of $c'_{✘}$ can be removed from $c'_{✘}$ to make the failure disappear or added to $c'_{✔}$ to make the failure occur.

The *dd* algorithm is defined as *dd* $(c_{✔}, c_{✘}) =$ *dd* $'(c_{✔}, c_{✘}, 2)$ with

$$dd'(c'_{✔}, c'_{✘}, n) = \begin{cases} (c'_{✔}, c'_{✘}) & \text{if } |\Delta| = 1 \\ dd'(c'_{✘} \setminus \Delta_i, c'_{✘}, 2) & \text{if } \exists i \in \{1..n\} \times test\,(c'_{✘} \setminus \Delta_i) = ✔ \\ & \text{("some removal passes")} \\ dd'(c'_{✔}, c'_{✔} \cup \Delta_i, 2) & \text{if } \exists i \in \{1..n\} \times test\,(c'_{✔} \cup \Delta_i) = ✘ \\ & \text{("some addition fails")} \\ dd'\big(c'_{✔} \cup \Delta_i, c'_{✘}, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1..n\} \times test\,(c'_{✔} \cup \Delta_i) = ✔ \\ & \text{("some addition passes")} \\ dd'\big(c'_{✔}, c'_{✘} \setminus \Delta_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1..n\} \times test\,(c'_{✘} \setminus \Delta_i) = ✘ \\ & \text{("some removal fails")} \\ dd'\big(c'_{✔}, c'_{✘}, \min(2n, |\Delta|)\big) & \text{else if } n < |\Delta| \text{ ("increase granularity")} \\ (c'_{✔}, c'_{✘}) & \text{otherwise} \end{cases}$$

where $\Delta = c'_{✘} \setminus c'_{✔} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$ with all Δ_i pairwise disjoint, and $\forall \Delta_i \times |\Delta_i| \approx (|\Delta|/n)$ holds.
The recursion invariant for *dd* $'$ is *test* $(c'_{✔}) = ✔ \wedge$ *test* $(c'_{✘}) = ✘ \wedge n \leqslant |\Delta|$.

dd algorithm returns a pair of configurations $c'_{\text{✘}}, c'_{\checkmark}$ that both lie between the original $c_{\checkmark}$ and $c_{\text{✘}}$ and the difference of which $\Delta = c'_{\text{✘}} \setminus c'_{\checkmark}$ is *1-minimal*—that is, each difference in Δ is relevant for producing the failure (see Proposition A.17 in the Appendix). Although in practice Δ frequently contains only one difference, Δ may contain multiple differences, which must *all* be applied to $c'_{\checkmark}$ in order to produce $c'_{\text{✘}}$.

Regarding complexity, *dd* has the same worst-case complexity as *ddmin*. If nearly all test cases result in an unresolved outcome, the number of tests can be quadratic with respect to $|c_{\text{✘}} \setminus c_{\checkmark}|$. The more test cases that pass or fail, though, the more efficient *dd* becomes, up to logarithmic complexity when all tests have a resolved outcome (see Proposition A.19 in the Appendix). When using *dd*, it is thus wise to keep unresolved test outcomes to a minimum, as this keeps down the number of tests required.

The *dd* algorithm can be seen as an *automation of scientific method*. It defines hypotheses (configurations), tests them, and refines or rejects the hypothesis according to the test outcome. One might argue that humans may be far more creative than this simple strategy. However, automating the process makes it less error prone and more systematic than most humans proceed—and it is exactly the type of boring task computers are supposed to relieve us from.

13.4 IMPLEMENTING ISOLATION

Example 13.3 shows a PYTHON implementation of *dd*. Just as with the PYTHON implementation of *ddmin* (Example 5.4), tail recursion and quantifiers have been turned into loops. Again, we rely on a `split()` function as well as set operations on lists such as `listunion()` (Example 13.4) and `listminus()`. (An implementation for `listunion()` is shown in Example 5.7.) Of course, we also need a `test()` function that returns either `PASS`, `FAIL`, or `UNRESOLVED` (e.g., a `test()` function as in Example 5.8 for MOZILLA). Extending the abstract form shown in Example 13.2, the concrete implementation has some fine points that reduce the number of tests:

- The order in which the test cases are checked is optimized. In particular, the first two cases require testing of `test(next_c_fail)` but not of `test(next_c_pass)`. This ensures a minimum number of tests, especially in cases where few tests are unresolved.
- We first check those situations that reduce the difference the most, such as cases 1–3.
- In principle, case 1 is not necessary, as it is subsumed by case 4. If successful, though, it avoids invoking `test(next_c_pass)`. Together, cases 1 and 2 turn `dd()` into a binary search if all tests return `PASS` or `FAIL` (rather than `UNRESOLVED`).
- The implementation assumes caching of earlier test results (see Section 5.8.1 in Chapter 5). If the `test()` function does not cache, you must rewrite the

code shown in Example 13.3 such that it saves and reuses the results of `test(next_c_pass)` and `test(next_c_fail)`.

- The `offset` variable records the subset to check next. When some difference becomes irrelevant (cases 4 and 5), we continue checking the next subset rather than restarting with the first subset. This makes sure each delta has the same chance to be removed.

EXAMPLE 13.3: A PYTHON implementation of the *dd* algorithm

```
def dd(c_pass, c_fail, test):
    """Return a pair (C_PASS', C_FAIL') such that
       * C_PASS subseteq C_PASS' subset C_FAIL' subseteq C_FAIL holds
       * C_FAIL' - C_PASS' is a minimal difference relevant for TEST."""
    n = 2     # Initial granularity

    while 1:
        assert test(c_pass) == PASS
        assert test(c_fail) == FAIL

        delta = listminus(c_fail, c_pass)
        if n > len(delta):
            return (c_pass, c_fail) # No further minimizing

        deltas = split(delta, n); assert len(deltas) == n

        offset = 0
        j = 0
        while j < n:
            i = (j + offset) % n
            next_c_pass = listunion(c_pass, deltas[i])
            next_c_fail = listminus(c_fail, deltas[i])

            if test(next_c_fail) == FAIL and n == 2:    # (1)
                c_fail = next_c_fail
                n = 2; offset = 0; break
            elif test(next_c_fail) == PASS:             # (2)
                c_pass = next_c_fail
                n = 2; offset = 0; break
            elif test(next_c_pass) == FAIL:             # (3)
                c_fail = next_c_pass
                n = 2; offset = 0; break
            elif test(next_c_fail) == FAIL:             # (4)
                c_fail = next_c_fail
                n = max(n - 1, 2); offset = i; break
            elif test(next_c_pass) == PASS:             # (5)
                c_pass = next_c_pass
                n = max(n - 1, 2); offset = i; break
            else:
                j = j + 1                      # Try next subset

        if j >= n:                             # All tests unresolved
            if n >= len(delta):
                return (c_pass, c_fail)
            else:
                n = min(n * 2, len(delta))    # Increase granularity
```

EXAMPLE 13.4: A PYTHON implementation of the `listunion()` function

```
def listunion(c1, c2):
    """Return the union of C1 and C2.
       Assumes elements of C1 are hashable."""

    # The hash map S1 has an entry for each element in C1
    s1 = {}
    for delta in c1:
        s1[delta] = 1

    # Add all elements in C2 that are not in C1
    c = c1[:]    # Copy C1
    for delta in c2:
        if not s1.has_key(delta):
            c.append(delta)

    return c
```

- Ordering of cases 3 and 4 is tricky. Case 4 simplifies the failing configuration and can rely on the result of a test already performed. Case 3 requires another test, but quickly reduces the difference if successful. Because we want to minimize the difference as quickly as possible, case 3 comes first.

Just like the PYTHON code for *ddmin* (List 5.2), this code should be easy to port to other languages. All you need is an appropriate representation for sets of circumstances such as `c_pass` or `c_fail`.

13.5 ISOLATING FAILURE-INDUCING INPUT

Let's now put *dd* to practice, applying it on a number of failure-inducing circumstances. We have already seen how *dd* pinpoints the < character in the HTML input (Example 13.1) and how this requires much fewer tests than simplifying the entire input (Example 5.5).

Applying *dd* on the fuzz inputs (discussed in Section 5.7 in Chapter 5) yields even more substantial savings. As reported in Zeller and Hildebrandt (2002), only 12–50 tests were required to narrow down the failure-inducing difference to a single character. This confirms the prediction of Proposition A.19 predicting a logarithmic number of tests when all tests have a resolved outcome. In case of the FLEX tests, where *ddmin* requires 11,000–17,960 test runs to simplify the input, the *dd* algorithm requires but 23–51 runs.

In cases where there were unresolved outcomes, as well as larger failure-inducing differences, the number of tests performed by *dd* was larger. In one of the NROFF test cases, 473 test runs (out of which 390 were unresolved) were needed to isolate a

17-character failure-inducing difference. However, this is still a much lower number than the 5,565 test runs required for simplification of the same input.

13.6 ISOLATING FAILURE-INDUCING SCHEDULES

Again, we can apply isolation on all circumstances that influence the program execution—provided we have a means of controlling and reproducing them. As discussed in Section 4.3.7 in Chapter 4, *schedules* of process and threads can result in failures that are difficult to debug. With a means of *recording and replaying* schedules, and a means of isolating failure-inducing differences, such defects become far easier to track down. The basic idea uses four building blocks:

Deterministic replay. Use a tool that captures the execution of nondeterministic JAVA applications and allows the programmer to *replay* these executions deterministically—that is, input and thread schedules are reconstructed from the recorded execution. This effectively solves the problem of reproducing failures deterministically.

Test case generation. A replay tool allows the application to be executed under a given thread schedule. Use the tool to generate *alternate schedules*. For instance, one can alter an original passing (or failing) schedule until an alternate failing (passing) schedule is found.

Isolating failure causes. Use *dd* to automatically isolate the failure cause in a failure-inducing thread schedule. The basic idea is to systematically *narrow the difference* between the passing and the failing thread schedule until only a minimal difference remains—a difference such as "The failure occurs if and only if thread switch #3291 occurs at clock time 47,539." This effectively solves the isolation problem.

Relating causes to errors. Each of the resulting thread differences occurs at a specific location of the program—for instance, thread switch #3291 may occur at line 20 of *foo.java*—giving a good starting point for locating thread interferences.

Choi and Zeller (2002) implemented this idea using IBM's DEJAVU tool to record and replay thread schedules on a single-processor machine. As a proof of concept, they applied the approach on a multithreaded ray-tracing program from the SPEC JVM98 JAVA test suite, in which they had reintroduced a *data race* that had been commented out by the original authors. This defect, shown in Example 13.5, lead to a failure the first time it was executed.

1. Thread *A* enters the `LoadScene()` method and saves the value of `ScenesLoaded` in `OldScenesLoaded` (line 84).
2. In line 85, a thread switch occurs, causing the transfer of control from thread *A* to another thread *B*.

EXAMPLE 13.5: Introducing a race condition

```
 25 public class Scene { ...
 44     private static int ScenesLoaded = 0;
 45     (more methods ...)
 81     private
 82     int LoadScene(String filename) {
 84         int OldScenesLoaded = ScenesLoaded;
 85         (more initializations ...)
 91         infile = new DataInputStream(...);
 92         (more code ...)
130         ScenesLoaded = OldScenesLoaded + 1;
131         System.out.println("" +
                ScenesLoaded + " scenes loaded.");
132        ...
134     }
135     ...
733 }
```

Source: Choi and Zeller (2002). `ScenesLoaded` may not be properly updated if a thread switch occurs during execution of lines 85–130.

3. Thread *B* runs the entire `LoadScene()` method and properly increments the `ScenesLoaded` variable.
4. As thread *A* resumes execution, it assigns the value of `OldScenesLoaded` plus one to `ScenesLoaded` (line 130). This effectively undoes the update made by thread *B*.

Using a fuzz approach (see Section 5.7 in Chapter 5), Choi and Zeller generated random schedules, starting from the failing one. Each schedule consisted of long lists of *yield points*—places in the program code such as function calls or backward branches where a thread switch occurred. After 66 tests, they had generated an alternate schedule where the failure would not occur.

Comparing the original (failing) schedule and the alternate (passing) schedule resulted in 3,842,577,240 differences, each moving a thread switch by one yield point. Applying all differences to the passing schedule changed its yield points to those in the failing schedule, thus making the program fail. However, only a few of these 3.8 billion schedule differences were relevant for the failure, which could be uncovered by delta debugging.

The delta debugging run is summarized in Figure 13.2. The upper line is the size of the failing configuration $c'_{✗}$, and the lower line is the size of the passing configuration $c'_{✓}$. As the tests only return ✓ or ✗, *dd* requires a logarithmic number of tests such that after 50 tests only one difference remains. The failure occurs if and only if thread switch #33 occurs at yield point 59,772,127 (instead of 59,772,126)—that is, at line 91 of *Scene.java*.

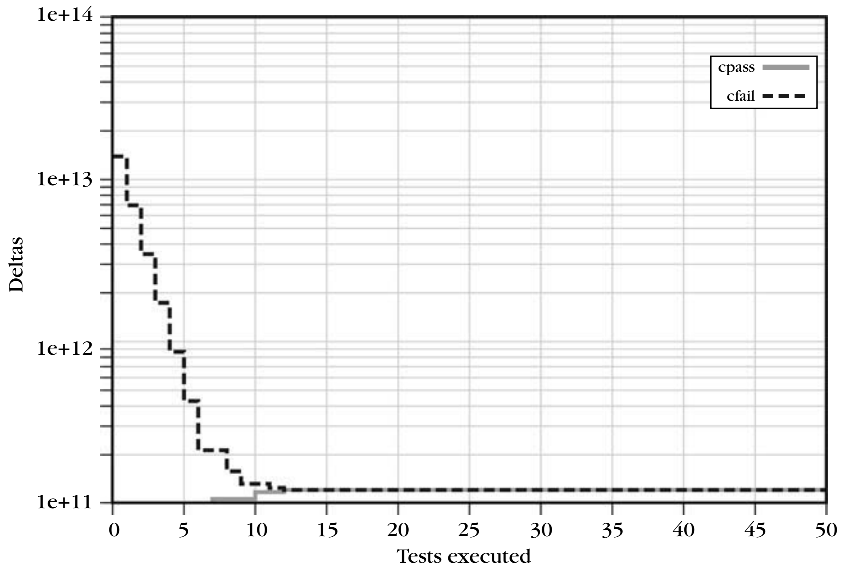


FIGURE 13.2

Narrowing down a failure-inducing thread switch. After 50 tests, one out of 3.8 billion thread switches is isolated as an actual failure cause.

Line 91 of *Scene.java* is the first method invocation (and thus yield point) after the initialization of *OldScenesLoaded*. Likewise, the alternative yield point 59,772,126 (with a successful test outcome) is the invocation of *LoadScene* at line 82 of *Scene.java*—just *before* the variable *OldScenesLoaded* is initialized. Thus, by narrowing down the failure-inducing schedule difference to one single difference the approach had successfully rediscovered the location where Choi and Zeller had originally introduced the error.

As this example was artificially generated, it does not necessarily generalize to all types of parallel programs. However, it illustrates that once we have a means of automated deterministic testing (as with the DEJAVU tool) adding automated isolation of failure-inducing circumstances is easy. In other words, once one has automated testing, automated isolation of failure causes is a minor step.

13.7 ISOLATING FAILURE-INDUCING CHANGES

Failure-inducing inputs and thread schedules do not directly cause a failure. Instead, they cause different executions of the program, which in turn cause the failure. An interesting aspect of thread switches is that they can be *directly associated with* *code*—the code executed at the moment the thread switch occurs (and is thus a failure cause). Consequently, the programmer can immediately focus on this part of the program code.

This is close to what we'd actually want: some machine where we can simply shove in our program and it will tell us "This line is wrong; please fix it" (or better yet, fix it for us such that we do not have to do any work at all anymore).

As surprising as it seems, such an approach exists, and it is based on delta debugging. It works as follows. Rather than having two different inputs for the same program, we have one input for two versions of the program—one version where the test passes, and one version where it fails. The goal of delta debugging is now to isolate the *failure-inducing difference* between the two versions—the change that turns a failing version into a passing version.

At this point, you may ask "Where should I get the passing version from? Is not this the whole point of debugging?" And you are right. However, there are situations in which some "old" version of a program passed a test that a "new" version fails. This situation is called a *regression*. The new version falls behind the capabilities of the old version.

The following is an example of a regression. From 1997 to 2001, I was maintaining the DDD debugger discussed in Section 8.6 in Chapter 8. DDD is a front end to the GDB command-line debugger, sending commands to GDB and interpreting its replies. In 1998, I got an email from a user who had upgraded his GDB version and suddenly DDD no longer worked properly.

```
Date: Fri, 31 Jul 1998 15:11:05 -0500
From: ⟨Name withheld⟩
To: DDD Bug Reports <bug-ddd@gnu.org>
Subject: Problem with DDD  and GDB  4.17

When using DDD  with GDB  4.16, the run command correctly
uses any prior command-line arguments, or the value of
"set args".  However, when I switched to GDB  4.17, this
no longer worked:  If I entered a run command in the
console window, the prior command-line options would be
lost. [...]
```

This regression situation is all too common when upgrading your system. You upgrade one part and suddenly other parts that depended on the "old" behavior no longer work. I wondered whether there was a way of isolating the cause automatically—that is, of isolating the change to GDB that caused DDD's failure.

If a regression occurs, a common debugging strategy is to focus on the *changes* one made. In our case, the change was a move from GDB 4.16 to GDB 4.17, and thus this part was clear. However, this change in the GDB release translates into several changes to the GDB source code. Running the `diff` utility to highlight those changes revealed an output of 178,200 lines.

```
$ diff -r gdb-4.16 gdb-4.17
diff -r gdb-4.16/COPYING gdb-4.17/COPYING
5c5
<                            675 Mass Ave, Cambridge, MA 02139, USA
---
>                         59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
282c282
<       Appendix: How to Apply These Terms to Your New Programs
---
>           How to Apply These Terms to Your New Programs
⋮
⟨178,192 more lines⟩
```

These 178,200 lines translate into 8,721 individual changes separated by unchanged lines—that is, there were 8,721 locations in the GDB source code that had been changed. At least one of these 8,721 changes caused the regression—but which one?

Again, this is a setting in which delta debugging can isolate the failure cause. The idea is to treat these changes as *input* to a *patch-and-test* program that works in three steps.

1. *Apply changes.* We must apply the changes to the GDB code base. This is done by taking the original-GDB -4.16 code base and then running the UNIX PATCH program to apply a subset of the changes. Note that PATCH may fail to apply the changes—for instance, if individual changes depend on each other. In this case, the test outcome is unresolved (**?**).
2. *Reconstruct GDB.* We must reconstruct GDB after all changes have been applied. Normally, this would be a simple matter of invoking the UNIX MAKE program. However, as the MAKE specification itself may have changed we need to recreate the *Makefile* first.

   If we apply a huge set of unrelated changes, we are quite likely to get a compilation error. The *patch-and-test* program must detect this and return an unresolved test outcome.
3. *Run the test.* If we have been successful in recreating GDB, we run it (with DDD) to see whether the failure occurs or not. Because applying arbitrary subsets of changes can result in surprising behavior of the program, it is wise to limit unwanted effects. In the case of GDB, we created a temporary directory for each run, ensuring that personal files would not be touched or overwritten.

Having translated the changes to input, we can now apply delta debugging to minimize the set of changes or to isolate the failure-inducing change (the failure-inducing input). The *patch-and-test* program, instrumented by *ddmin* or *dd*, would apply a subset of the changes, see whether GDB can be reconstructed, and if so return ✔ or ✘ depending on the test outcome. If GDB cannot be reconstructed with the changes applied (which is quite common for random subsets), the *patch-and-test* program would return **?**, and delta debugging would try the next alternative. If we actually do this, and run delta debugging, we end up in a single change that makes DDD fail.

```
diff -r gdb-4.16/gdb/infcmd.c gdb-4.17/gdb/infcmd.c
1239c1278
< "Set arguments to give program being debugged when it is started.\n\
---
> "Set argument list to give program being debugged when it is started.\n\
```

This change in a string constant from `arguments` to `argument list` was responsible for GDB 4.17 not interoperating with DDD. Although the string constant is actually part of GDB's online help, it is also the basis for GDB's output. Given the command `show args`, GDB 4.16 replies

```
Arguments to give program being debugged when it is started is "a b c"
```

but GDB 4.17 issues a slightly different (and grammatically correct) text:

```
Argument list to give program being debugged when it is started is "a b c"
```

Unfortunately, this output could not be parsed by DDD, which expected a reply starting with "Arguments." To solve the problem here and now, one could simply have reversed the GDB change. Eventually, I upgraded DDD to make it work with the new-GDB-version.

This approach of determining the culprit for a regression has been named the *blame-o-meter*—as a means to know who to blame. However, as the GDB example shows, a *cause* for a problem need not be a defect, it may not even be a mistake. What the GDB programmers did was perfectly sensible. DDD's defect, if any, was to rely on a specific output format. Nonetheless, once one has an automated regression test it may prove useful to add a blame-o-meter on top. This way, whenever a regression test fails one could start the blame-o-meter and tell the developer not only *that* a test fails but also *why* it fails.

Building a blame-o-meter is not very difficult, provided one has automated construction, automated regression tests, and a means of applying changes (such as the UNIX PATCH program). A number of issues call for specific *optimizations*, though.

History. If the changes come from a version archive, they can be grouped according to their creation time. Ensuring that later changes always occur with earlier changes will speed up delta debugging enormously, as this ensures consistent reconstruction and thus resolved test outcomes. In addition, as we know from Proposition A.19 in the Appendix, resolved tests outcomes result in a logarithmic number of tests—basically a binary search along the change history.

Reconstruction. As each test requires reconstruction of the program, it is useful to have a means of *incremental* reconstruction. The MAKE program compiles only the code the sources of which have changed since the last construction. The CCACHE program speeds up recompilation by caching previous compiles and detecting when the same compile is being done again.

Grouping. Many subsets of changes result in unresolved outcomes because the program cannot be reconstructed. For instance, a change *A* that references a variable may require a change *B* that declares that variable. Every subset that contains only *A* but not *B* will result in an unresolved outcome, slowing down delta debugging considerably. Therefore, in addition to grouping changes by creation time, it is also useful to group them according to *scope*—that is, to keep those changes together that apply to the same file, class, or function.

Failure resolution. A simple means of dealing with construction errors is to search for changes that may fix these errors. After a failing construction, one could scan the error messages for identifiers, add all changes that reference these identifiers, and try again. This is repeated until construction is possible, or until there are no more changes to add.

In the case of the 8,721 changes to the GDB source code, these optimizations all proved beneficial. CCACHE reduced the average reconstruction time to 20 percent. Grouping by scope reduced the number of tests by 50 percent. Overall, as Figure 13.3 shows, using *dd* with scope information required about 97 tests. Assuming that each test takes about two minutes, this boils down to three hours until delta debugging has isolated the cause in the GDB code.

Three hours still sounds like a lot. I am pretty confident that you as a programmer would have found the cause in the GDB code in less than three hours (especially having read this book). However, it is rather uncommon to have 8,721 changes without any temporal ordering and intermediate regression tests, such that typical regressions can be dealt with much faster.

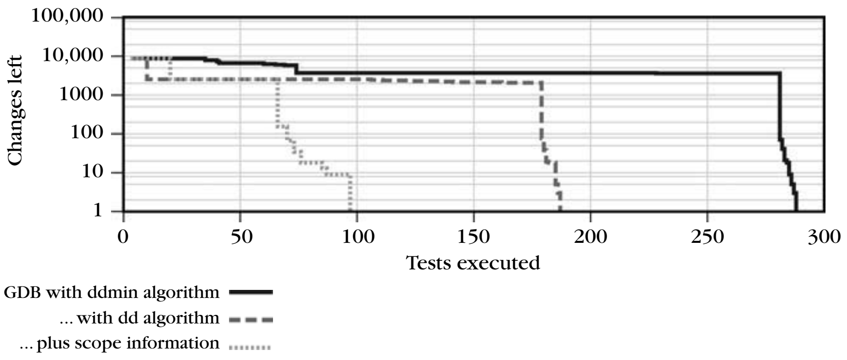
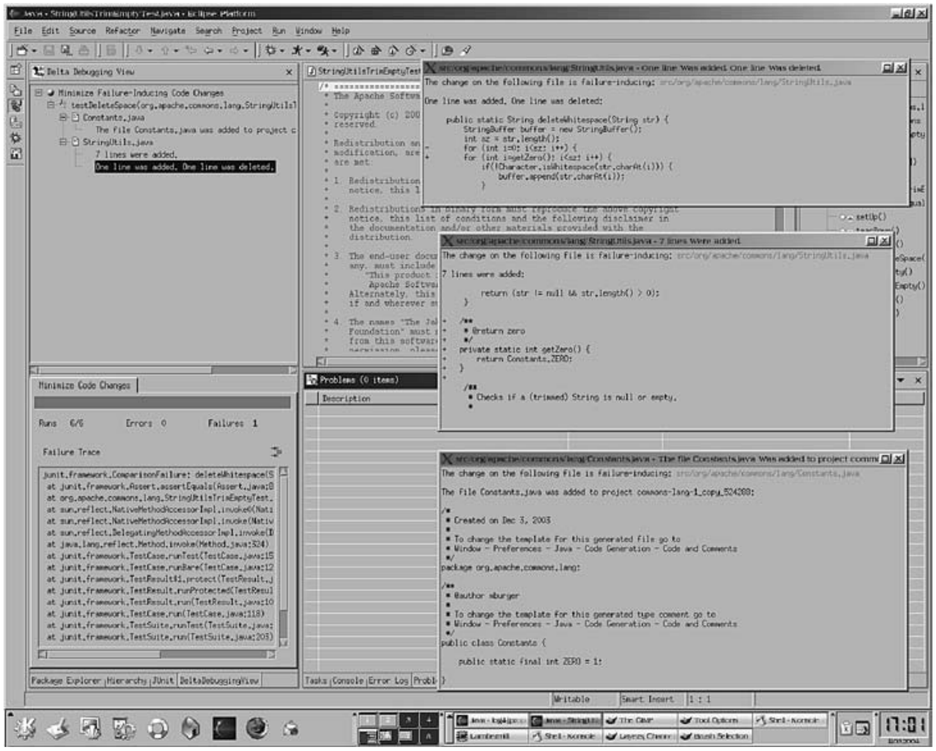


FIGURE 13.3

Isolating failure-inducing code changes. After 97 tests, delta debugging has isolated one of 8,721 code changes as causing the failure.



```
Delta Debugging View
Minimize Failure-Inducing Code Changes
  testDeleteSpace(org.apache.commons.lang.StringUtilsT
    Constants.java
      The file Constants.java was added to project c
    StringUtils.java
      7 lines were added.
      One line was added. One line was deleted.
```

```
Minimize Code Changes
Runs  6/6     Errors  0     Failures  1

Failure Trace
junit.framework.ComparisonFailure: deleteWhitespace(S
 at junit.framework.Assert.assertEquals(Assert.java:8
 at org.apache.commons.lang.StringUtilsTrimEmptyTest.
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Nati
 at sun.reflect.NativeMethodAccessorImpl.invoke(Nativ
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(D
 at java.lang.reflect.Method.invoke(Method.java:324)
 at junit.framework.TestCase.runTest(TestCase.java:15
 at junit.framework.TestCase.runBare(TestCase.java:12
 at junit.framework.TestResult$1.protect(TestResult.j
 at junit.framework.TestResult.runProtected(TestResul
 at junit.framework.TestResult.run(TestResult.java:10
 at junit.framework.TestCase.run(TestCase.java:118)
 at junit.framework.TestSuite.runTest(TestSuite.java:
 at junit.framework.TestSuite.run(TestSuite.java:203)
```

```
src/org/apache/commons/lang/StringUtils.java - One line was added. One line was deleted.
The change on the following file is failure-inducing: src/org/apache/commons/lang/StringUtils.java

One line was added. One line was deleted:

    public static String deleteWhitespace(String str) {
        StringBuffer buffer = new StringBuffer();
        int sz = str.length();
-       for (int i=0; i<sz; i++) {
+       for (int i=getZero(); i<sz; i++) {
            if(!Character.isWhitespace(str.charAt(i))) {
                buffer.append(str.charAt(i));
            }
```

```
src/org/apache/commons/lang/StringUtils.java - 7 lines were added
The change on the following file is failure-inducing: src/org/apache/commons/lang/StringUtils.java

7 lines were added:

        return (str != null && str.length() > 0);
    }

+   /**
+    * @return zero
+    */
+   private static int getZero() {
+       return Constants.ZERO;
+   }
+
    /**
     * Checks if a (trimmed) String is null or empty.
     *
```

```
src/org/apache/commons/lang/Constants.java - The file Constants.java was added to project comm
The change on the following file is failure-inducing: src/org/apache/commons/lang/Constants.java

The File Constants.java was added to project commons-lang-1_copy_524288:

/*
 * Created on Dec 3, 2003
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package org.apache.commons.lang;

/**
 * @author mburger
 *
 * To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
public class Constants {

    public static final int ZERO = 1;

}
```

FIGURE 13.4

Isolating failure-inducing code changes in ECLIPSE. As soon as a test fails, the delta debugging plug-in automatically determines the failure-inducing code change—if there were an earlier version where the test did not fail.

If you want to experiment with delta debugging, Figure 13.4 shows a plug-in named DDCHANGE for the ECLIPSE programming environment. DDCHANGE keeps track of all tests and all test outcomes. As soon as a test fails that had passed in some previous version, the delta debugging plug-in automatically determines the failure-inducing code change in the background and presents its diagnosis as soon as it is available. This plug-in makes use of the facilities for automated construction, automated testing, and version control, as they are integrated within ECLIPSE (thus, no complex setup is necessary).

DDCHANGE is not that quick, either. As it has to reconstruct the program several times, it can take some time to isolate the failure-inducing change—and again, an experienced programmer might find the cause quicker. However, unless you *need* the cause in the next hour you can easily have delta debugging determine the cause for you. It is slow and dumb, but it will come up with a result—and no programmer I know of has fun running these experiments manually.

13.8 PROBLEMS AND LIMITATIONS

Although delta debugging is generally good at isolating causes, one should be aware of its limits as well as common issues. These include the following

How do we choose the alternate world?

As laid out in Section 12.7 in Chapter 12, choosing the alternate world (i.e., input or version) determines the *initial difference* in which to search for causes. If we want to search for causes in the input, we should use a passing run with a different input. If we want to search for causes in the configuration, we should search a passing run with an alternate configuration. In general, we should choose an alternate world that is as close as possible to the actual world, in order to keep the initial search space as small as possible.

How do we decompose configurations?

Many configurations naturally decompose into smaller parts that can be individually assessed. A text decomposes into lines, an HTML page decomposes into head and body sections, a schedule decomposes into thread switches, and a code difference decomposes into locations. However, there are some configurations for which it is difficult to provide such a decomposition.

As an example, consider *image processing*—wherein an application fails when processing one specific image but passes on all others. What is the difference between the failure-inducing image and the passing ones? In such cases, it helps to understand how the application works. Does it process the image row by row? In such cases, decomposing the image by rows may make sense. Does it rely on properties such as number of colors, brightness, or contrast? In this case, it may make sense to reduce the difference in terms of these properties—for instance, to have each delta adjust the contrast until the difference becomes minimal.

When do we know a *failure is* the *failure?*

When a program is fed with arbitrary input, it may fail in a number of ways. However, the changed input may cause a *different failure* than the original test case—that is, the program may fail at a different location, produce an alternate error message, or otherwise produce a behavior that is considered "failing" but differs from the original behavior. We call such different failures *artifacts*, which are *artificially generated* during the delta debugging process.

In the fuzz examples from Section 5.7 in Chapter 5, for instance, our *test* function would return ✗ whenever a program crashed—regardless of further circumstances. In the fuzz case, ignoring artifacts may be legitimate, as a program must not crash under any input. In general, though, we may want to check further aspects of the failing behavior.

One of the most important aspects about a failure is to know the location that was active at the moment of the failure—that is, the exact statement that issued the failing message, or the last executed statement in case of a crash. Checking this location is a good protection against artifacts. An even better protection is to take into account the backtrace, the stack of calling functions (see Section 8.3.1 in Chapter 8) at the time of the failure.

- The *test* function returns ✘ only if the program failed and if the backtrace of the failure was identical to the original backtrace.
- If the program failed, but with a different backtrace, *test* would return **?**.
- If the program passed the test, *test* would return ✔.

In addition to the backtrace, further aspects such as coverage, timing, or exact output can be used to check against artifacts. However, the larger the number of aspects to be considered part of a failure, the larger the cause required to create all of these aspects.

How do we disambiguate between multiple causes?

For the sake of efficiency, delta debugging always takes the first possibility to narrow down a difference. However, there may be other possibilities, resulting in alternate actual causes. In the MOZILLA example, for instance, *dd* returned `<` as a failure-inducing difference, but removing any of the `<SELECT>` characters would have made the failure disappear.

It is fairly easy to extend *dd* such that it considers other alternatives. This is a trade-off between performance and convenience. In my experience, the first cause is typically sufficient to characterize the failure. If it were not, I would run *dd* on the other alternatives. Others may prefer to have *dd* compute multiple alternatives in advance, such that they can consider them all.

How do I get to the defect?

Every failure-inducing difference returned by delta debugging is an actual cause. As such, it suggests a *fix*: simply remove the cause and the failure will no longer occur. For instance, we could remove the `<` from the MOZILLA input, prohibit thread switches during the Raytracer data race, or revert the GDB code to the previous version. This illustrates that such fixes are more *workarounds* than corrections—simply because they do not increase correctness of the program.

In general, though, the *cause* delta debugging isolates is seldom an *error*—simply because the alternate world need not be correct, either. In fact, if an error is the same in both worlds it will not even be part of the difference in which delta debugging searches for an actual cause.

To turn the cause into a *correction*, we still have to find out where to correct the program (which is, in fact, *deciding* where and what the defect is). We would have to search the code that handles printing of `<SELECT>` tags in MOZILLA, set up the Raytracer such that the data race no longer occurs, and adapt DDD such that it handles the outputs from different GDB versions.

In all three cases, the correction is induced by the cause, and the cause certainly helps in designing and motivating the correction. To get to the actual defect, though, again requires us to examine the innards of the program, as explored in the remainder of this book.

13.9 CONCEPTS

To isolate failure causes automatically, you need

- an *automated test* that checks whether the failure is still present,
- a means of *narrowing down the difference*, and
- a *strategy* for proceeding.

One possible strategy is the general delta debugging algorithm *dd* (Example 13.2).

dd determines the *relevant difference* between two configurations (inputs, schedules, code changes, or other circumstances) with respect to a given test—that is, an actual cause for the failure.

How To

To isolate a failure cause in the input, apply *dd* (or another strategy) on two program inputs—one that passes and one that fails the test.

To isolate a failure cause in the thread schedule, apply *dd* (or another strategy) on two schedules—one that passes and one that fails the test. You need a means of replaying and manipulating schedules, such as DEJAVU.

To isolate a failure-inducing code change, apply *dd* (or another strategy) on two program versions—one that passes and one that fails the test. You need automated reconstruction of the program after a set of changes has been applied.

Any actual cause, as returned by delta debugging, can be altered to make the failure no longer occur. This does not mean, though, that the cause is a defect. It also does not mean that there may be only one actual cause.

Delta debugging on states is a fairly recent technique and not yet fully evaluated.

13.10 TOOLS

Delta Debugging Plug-ins for ECLIPSE. At the time of writing, a number of delta debugging tools were made available for the ECLIPSE programming framework. They can be downloaded at *http://www.st.cs.uni-saarland.de/eclipse/*.

CCACHE. To apply delta debugging on program changes, you may find the CCACHE tool for incremental compilation useful. It is available at *http://ccache.samba.org/*.

13.11 FURTHER READING

Delta debugging on program inputs is described in Zeller and Hildebrandt (2002), a paper discussed in Chapter 5. The paper cites all data from all experiments. The definitions are general enough to pertain to all types of changes and configurations.

Delta debugging on thread schedules was developed by Choi and Zeller (2002) while the authors were visiting IBM research. This paper contains all details on the approach and the experiment. The DEJAVU tool by Choi and Srinivasan (1998) is described in Chapter 4.

Zeller (1999) describes how to apply delta debugging to code changes. This was the first application of delta debugging. The algorithms used in this paper are now superseded by the more advanced versions in this book, but the case studies are still valid.

Failure-inducing code changes were first handled by Ness and Ngo (1997). In their setting, a compiler consisted of a number of optimization modules. By reverting module after module to an earlier (passing) state, they succeeded in identifying the module the changes of which caused the failure and therefore kept it at its earlier revision.

EXERCISES

13.1 Repeat the exercises of Chapter 5 "Simplifying Problems" using isolation instead of minimization.

13.2 Implement simplification of test cases using an unchanged *dd* implementation, but with a wrapper around the *test* function. Which wrapper is needed?

13.3 Rather than simplifying the failing configuration (as in *ddmin*), one can also think about *maximizing the passing configuration*—that is, having the largest possible configuration that still passes (with a minimal difference to the failing configuration).

(a) When would such a *ddmax* algorithm be useful?

(b) Give a mathematical description of *ddmax* (analogously to *ddmin* in List 5.2).

(c) Implement *ddmax*.

13.4 Each statement about causes and effects is either true or false. Give a short reason for your answer if appropriate.

- If C is a cause and E is its effect, C must precede E. T/F
- If C is a circumstance that causes a failure, it is possible to alter C such that the failure no longer occurs. T/F
- If some cause C is an actual cause, altering C induces the smallest possible difference in the effect. T/F
- Every failure cause implies a possible fix. T/F
- For every failure, there is exactly one actual cause. T/F
- A failure cause can be determined without executing a program. T/F
- If I observe two runs (one passing, one failing) with a minimal difference in input, I have found an actual failure cause. T/F
- A successful fix proves that the altered code was the actual failure cause. T/F

13.5 In delta debugging, you can either use simplification to simplify failure-inducing input or isolation to isolate a minimal failure-inducing difference.

(a) Compare these two approaches with respect to their advantages and disadvantages.

(b) Compare the running times of the respective algorithms in their worst-case and best-case behavior.

(c) Which are the properties of a 1-minimal result of isolation?

13.6 Using the logbook format (Section 6.5 in Chapter 6), describe the first four steps of the delta debugging run in Example 13.1. Which are the hypotheses, predictions, and experiments?

13.7 Is delta debugging an instance of scientific method? Discuss.

13.8 Which are the prerequisites in order to apply delta debugging? Discuss situations in which delta debugging is not helpful.

Debugging is still, as it was 30 years ago, a matter of trial and error.

– Henry Lieberman
The Debugging Scandal (1997)