CHAPTER

Simplifying Problems 5

Once we have reproduced a problem, we must *simplify* it—that is, we must find out which circumstances are not relevant for the problem and can thus be omitted. This process results in a test case that contains only the relevant circumstances. In the best case, a simplified test case report immediately pinpoints the defect. In this chapter, we introduce *delta debugging*, an automated debugging method that simplifies test cases automatically.

5.1 SIMPLIFYING THE PROBLEM

After one has reproduced a problem, the next step in debugging is to find out what is relevant and what is not. Does the problem really depend on the entire 10,000 lines of input? Is it really necessary to replay all of these interaction steps? Does the failure occur only if this exact schedule of events takes place? Do I really need this long sequence of recorded method calls?

This stage of debugging is known as *simplifying,* meaning to turn a detailed problem report into a simple *test case.* A test case contains the *relevant* details only. A detail of the problem report is relevant if it is *required* to make the problem occur. It is *irrelevant* if the problem occurs whether the detail is present or not.

Why is simplification important? As an analogy, consider a simple *flight test*. An airplane crashes a few seconds after taking off. To find out what happened, we repeat the flight in a simulator.

Even if we do not have a clue about how planes work, we can still find out what is relevant and what is not, by repeating the flight over and over again under *changed circumstances.* For instance, we might take out the passenger seats and find that the plane still crashes. We might take out the coffee machine and the plane still crashes. We might take out the engines and—oops, the plane does not move off the runway. Obviously, the engines are important!

Eventually, only the relevant "simplified" skeleton remains, including a (simulated) test pilot, the wings, the runway, the fuel, and the engines. Each part of this skeleton is relevant for reproducing the crash.

To explain how the crash came to be, we need every single part of this skeleton. However, the value of simplification is less in the remaining parts but rather in all parts that have been taken away—all of the irrelevant details (such as the coffee machine that did not contribute to the crash. The general process for simplifying a problem report follows a simple rule:

> *For every circumstance of the problem, check whether it is relevant for the problem to occur. If it is not, remove it from the problem report or the test case in question.*

A *circumstance* is any aspect that may influence the problem—in short, the same circumstances one needs to reproduce the problem (see Chapter 4). In particular, these are:

- Aspects of the problem environment
- Individual steps in the problem history

How does one check whether a circumstance is relevant? You do this by *experimenting*. That is, you omit the circumstance and try to reproduce the problem. If the problem no longer occurs, the circumstance is relevant. If the problem still occurs, the circumstance is irrelevant. As McConnell (1993) puts it: "The goal of simplifying the test case is to make it so simple that changing any aspect of it changes the behavior of the error."

This is exactly our key question:

HOW DO WE SIMPLIFY TEST CASES SYSTEMATICALLY AND AUTOMATICALLY?

5.2 THE GECKO BUGATHON

Simplification of test cases is not an academic problem. Consider a real-world example, related to the MOZILLA Web browser—or more specifically, its HTML layout engine *Gecko*. In July 1999, two years before the final completion of MOZILLA 1.0, BUGZILLA (the MOZILLA problem database) listed database) listed more than 370 open problem reports—problem reports that were not even reproduced.

In Example 3.1 we have already seen one of these open problem reports, reported by a MOZILLA user in 1999. This problem report is already close to perfection: it is short, reproducible, and precise. It can also easily be automated, as discussed in Chapter 3. The problem, though, is that the Web page in question—the page at *http://bugzilla.mozilla.org*—was 896 lines of quite obfuscated HTML code (shown in Example 5.1). Loading this HTML code into Gecko HTML input made MOZILLA fail—but what? made MOZILLA fail—but what?

Obviously, reading this HTML code does not give us any hints about possible failure causes. If we were MOZILLA programmers, what we want here is the *simplest HTML input* that still produces the failure—and hopefully pinpoints the failure cause.

EXAMPLE 5.1: Printing this HTML page (excerpt) makes MOZILLA crash

```
<td align=left valign=top>
<SELECT NAME="op_sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION
VALUE="Windows 95">Windows 95<OPTION VALUE="Windows 98">Windows 98<OPTION
VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows
2000<OPTION VALUE="Windows NT">Windows NT<OPTION VALUE="Mac System 7">Mac
System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System
8.0<OPTION VALUE="Mac System 8.5">Mac System 8.5<OPTION VALUE="Mac
System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac System
9.x<OPTION VALUE="MacOS X">MacOS X<OPTION VALUE="Linux">Linux<OPTION
VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION
VALUE="NetBSD">NetBSD<OPTION VALUE="OpenBSD">OpenBSD<OPTION
VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION
VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION
VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION
VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION
VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="-">-<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION
VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug_severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION
VALUE="major">major<OPTION VALUE="normal">normal<OPTION
VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION
VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

A simplified test case not only helps in finding failure causes, though. There are at least three further good reasons for simplifying:

1. *A simplified test case is easier to communicate.* The simpler a test case, the less time it takes to write it down, to read its description, and to reproduce it. In addition, you know that the remaining details are all relevant because the irrelevant details have been taken away. In our example, is it relevant that the margins be set to `.50`? If the failure occurs nonetheless, we can leave out this detail.
2. *A simplified test case facilitates debugging.* Typically, a simplified test case means less input (and thus smaller program states to examine) and less interaction

with the environment (and thus shorter program runs to understand). Obviously, if the HTML code in Example 5.1 can be simplified to a small number of HTML tags, the state of Gecko would be much easier to examine. In the best case, the HTML tag could even directly lead to the error.

3. *Simplified test cases identify duplicate problem reports.* As discussed in Section 2.8 in Chapter 2, duplicate problem reports can fill up your problem database. Simplified test cases typically *subsume* several duplicate problem reports that differ only in irrelevant details. If we know that some specific HTML tag causes printing to fail, we can search for this HTML tag in other problem reports, marking them as duplicates.

Despite these benefits, new problem reports came in quicker than MOZILLA programmers could possibly simplify them or even look at them. With this queue growing further, the MOZILLA engineers "faced imminent doom."

But then, Eric Krock, MOZILLA product manager, had a clever idea: Why not have *volunteers* simplify test cases? Thus, Krock started what became the *Gecko BugAThon*: Volunteers would help the MOZILLA programmers by creating simplified test cases. To simplify test cases, you do not have to be a programmer. All you need is a *text editor* (as shown in List 5.1). The entire process boils down to removing parts of the page and periodically rerunning MOZILLA until all remaining input is relevant.

As an incentive, Krock offered *rewards* for simplified test cases. For 5 problem reports turned into simplified test cases, a volunteer would be invited to the launch party. For 10 test cases, he or she would also get an attractive *Gecko stuffed animal*, and 20 test cases would earn him or her a T-shirt signed by the grateful engineers. This simple scheme worked out very well; because of the large number of enthusiastic volunteers on the Web the very first night a number of volunteers earned their stuffed animal by staying up late and simplifying test cases.

LIST 5.1: Instructions for Simplifying HTML Pages Manually

- Download the Web page that shows the bug to your local machine.
- Using a text editor (such as Notepad on Windows, SimpleText on the Mac, or vi or emacs on UNIX), start removing HTML markup, CSS rules, and lines of JavaScript from the page. Start by commenting out parts of the page (using `<!-- -->`) that seem unrelated to the bug. Every few minutes, check the page to make sure it still reproduces the bug. Code not required to reproduce the bug can be safely removed.
- You will do well if you use an editor supporting multiple levels of Undo, and better if you use an HTML-editing tool that supports preview to an external browser.
- When you have cut away as much HTML, CSS, and JavaScript as you can—and cutting away any more causes the bug to disappear—you are done.

Source: mozilla.org.

5.3 MANUAL SIMPLIFICATION

How would a MOZILLA volunteer proceed in an actual example? Let's apply the instructions in List 5.1 on the HTML input in Example 5.1. We use a method sketched by Kernighan and Pike (1999):

> *Proceed by binary search. Throw away half the input and see if the output is still wrong; if not, go back to the previous state and discard the other half of the input.*

This *divide-and-conquer* process is sketched in Figure 5.1.

1. The gray bar stands for the HTML input—initially 896 lines that cause MOZILLA to fail (✗).
2. Using a text editor, we cut away the second half of the input (shown in light gray), leaving only the first half (dark gray), and repeat the test with this 418-line input. MOZILLA still crashes (✗).
3. Again, we cut away the second half, leaving only 224 lines. Still, MOZILLA crashes.
4. When we again cut away the second half, leaving only 112 lines, MOZILLA just works (✓).
5. We undo the earlier cut and cut away the first half instead. When being fed with these 112 lines, MOZILLA again crashes.
6. We continue simplifying the input.
7. After 12 tests, one single line with a `<SELECT>` tag is left:

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

   (This HTML line in *http://bugzilla.mozilla.org/* is used to have users input the *problem priority* of a report.)

We have now simplified the problem report from 896 lines to 1 single line. Further testing shows that the tag attributes are irrelevant, too, and thus all we need to cause the problem is an input of `<SELECT>`.

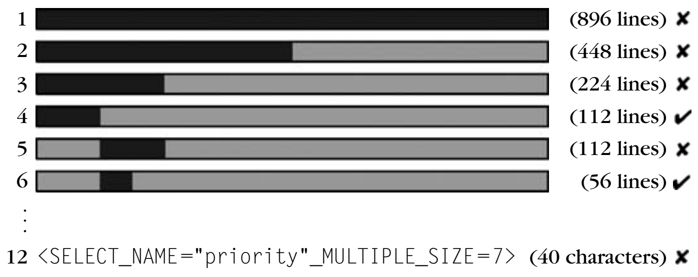

FIGURE 5.1

Simplifying the HTML input from Example 5.1.

Having simplified this problem is very beneficial. In particular, it helps in:

- *Communication:* All one needs is the three-word summary "Printing `<SELECT>` crashes."
- *Debugging:* A MOZILLA programmer can immediately focus to the piece of code that handles printing of `<SELECT>` tags.
- *Duplicates:* A Bugzilla maintainer can scan the database for other problems with printing, and if `<SELECT>` is part of the respective HTML input, chances are that they are all duplicates of each other.

5.4 AUTOMATIC SIMPLIFICATION

Manual simplification, as demonstrated in Section 5.3, has important benefits. However, these come at a price:

- *Simplification is tedious.* You have to run tests manually all over again.
- *Simplification is boring.* It is a rather mechanical activity without great intellectual challenge.

As with so many other tedious and boring activities, one may wonder whether it would not be possible to *automate* the simplification process. And indeed, it can. Once again, we illustrate the basic idea using the MOZILLA example:

- We set up an *automatic test* that determines whether MOZILLA fails to print on some specific input.
- We implement a *strategy* that realizes the binary search strategy mentioned earlier, running the test on some subset of the HTML page.

Setting up an automatic test for MOZILLA is not too difficult, applying the basic strategies from Chapter 4: We have MOZILLA read its input from file (rather than from the network), and use record/replay to automate the user interaction from Example 3.1. The test can thus be realized as follows:

1. Launch MOZILLA.
2. Use Capture and Replay to:
   - Load the HTML page into MOZILLA
   - Set printing settings as described in the problem report
   - Print the page
3. Wait for a certain amount of time to see whether:
   - MOZILLA crashes—that is, the test fails (✗)
   - Whether it is still alive—that is, the test passes (✓)
4. If MOZILLA should not start, or if it fails for some other reason, our test returns.

Let's consider the second part: to design an *automatic simplification strategy* using such an automatic test. As a starting point, we simply adapt the "binary search" strategy from Section 5.3:

1. Cut away half the input and check if the test returns ✘. If so, continue the process with the remaining half.
2. Otherwise, go back to the previous state and discard the other half of the input.

This simple description has a drawback: What do we do if *neither half* fails the test—that is, testing the first half passes and testing the second half passes as well? As an example, consider Example 5.2, where we attempt to simplify the remaining HTML line by *characters.* Again, input that has been cut away is shown in gray characters. Neither the first nor second half is valid HTML, and thus MOZILLA interprets the input as text and the test does not fail.

A simple binary search does not suffice any longer, in that we are not searching for a single character but for a *subset* of the input. In this example, the subset we are searching for is the string `<SELECT>`, spread across the two input halves.

How do we deal with this situation? The answer is not to cut away *halves* of the input, but *smaller parts*—quarters, for instance. Thus, instead of cutting away the first half, we cut away the first quarter, the second quarter, and so on.

This process is illustrated in Example 5.3, continuing the example from Example 5.2. Removing the first quarter (step 4, the first 10 characters) still does not cause the problem to occur, but removing the second quarter (step 5, characters 11–20) is successful. MOZILLA fails.

Now that we have a failure on a simplified input, should we go back to cutting away halves or should we continue with quarters? One could argue that we should at least test all subsets at the given granularity. Thus, we continue removing quarters (Step 6) until the last one (Step 7).

If removing neither quarter makes the test fail, we continue with eighths, and then sixteenths, and so on. Eventually, we will thus come down to a point where we remove single characters—and end up in an input where removing any character causes the problem to disappear (a single `<SELECT>` tag). Thus, our automatic simplification strategy has eventually cut away everything that is irrelevant for producing

EXAMPLE 5.2: Simplifying a single line

```
1   <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>   (40 characters)   ✘
2   <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>   (20 characters)   ✔
3   <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>   (20 characters)   ✔
```

EXAMPLE 5.3: Simplifying by quarters

```
4   <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>   (30 characters)   ✔
5   <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>   (30 characters)   ✘
6   <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>   (20 characters)   ✘
7   <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>   (10 characters)   ✔
```

the problem—and has done so only by trial and error, without any knowledge of the and error, without any knowledge of the program or the input.

5.5 A SIMPLIFICATION ALGORITHM

Let's now write down a general algorithm that realizes the automatic strategy sketched in Section 5.4. We have some test function *test*(c) that takes some input c and determines whether the failure in question occurs (✗, "fail") or not (✓, "pass") or whether something different happens (**?**, "unresolved").

Assume we have some failure-inducing input $c_✗$ that we can split into subsets. If we split $c_✗$ into two subsets c_1 and c_2, three things can happen:

- *Removing first half fails.* If *test* $(c_✗ \setminus c_1) = ✗$, we can continue with $c'_✗ = c_✗ \setminus c_1$. ($c'_✗$ is the value for $c_✗$ in the next iteration.)
- *Removing second half fails.* Otherwise, if *test* $(c_✗ \setminus c_2) = ✗$, we can continue with $c'_✗ = c_✗ \setminus c_2$.
- *Increase granularity.* Otherwise, we must increase the granularity and split $c_✗$ into four (eight, sixteen, and so on) subsets.

To accommodate the last case, we must generalize our description to an arbitrary number of subsets n. If we split $c_✗$ into n subsets c_1 to c_n, we get:

- *Some removal fails.* If *test* $(c_✗ \setminus c_i) = ✗$ holds for some $i \in \{1, \ldots, n\}$, continue with $c'_✗ = c_✗ \setminus c_i$ and $n' = \max(n-1, 2)$.
- *Increase granularity.* Otherwise, continue with $c_✗{}' = c_✗$ and $n' = 2n$. If $c_✗$ cannot be split into more subsets, we are done.

Let's generalize this further. Data input is just one way to determine a program's execution. In Chapter 4, we saw the influence of other circumstances such as time, communications, or thread schedules. We call such a set of circumstances that influence program behavior a *configuration.*

Our aim is now to find a *minimal set of circumstances* under which the failure occurs. That is, we want to *minimize a failure-inducing configuration* (to minimize the "failing" configuration $c_✗$). In the MOZILLA case, the HTML input is such a configuration—a set of circumstances that determine MOZILLA's execution—and we want to minimize this far as possible.

This generalization ends up in the *ddmin* algorithm shown in List 5.2, as proposed by Zeller and Hildebrandt (2002). Its core, the *ddmin*′ function, gets two arguments: the configuration (input) to be simplified (denoted as $c'_✗$) and the granularity n. Depending on test results, *ddmin*′ invokes itself recursively with a smaller $c'_✗$ ("some removal fails"), invokes itself recursively with double granularity ("increase granularity"), or ends the recursion.

ddmin is an instance of *delta debugging*—a general approach to isolate failure causes by narrowing down differences (*deltas*) between runs. (More precisely,

LIST 5.2: The *ddmin* Algorithm in a Nutshell

- Let a program's execution be determined by a set of circumstances called a *configuration.* The set of all circumstances is denoted by $\mathcal{C}$.
- Let *test*: $2^{\mathcal{C}} \rightarrow \{\text{✘}, \text{✔}, \textbf{?}\}$ 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 (**?**).
- Let $c_{\text{✘}}$ be a "failing" configuration with $c_{\text{✘}} \subseteq \mathcal{C}$ such that *test*$(c_{\text{✘}}) = \text{✘}$, and let the test pass if no circumstances are present [i.e., test $(\emptyset) = \text{✔}$].
- The *minimizing delta debugging algorithm ddmin*$(c_{\text{✘}})$ minimizes the failure-inducing configuration $c_{\text{✘}}$. It returns a configuration $c'_{\text{✘}} =$ *ddmin*$(c_{\text{✘}})$ such that $c'_{\text{✘}} \subseteq c_{\text{✘}}$ and *test*$(c'_{\text{✘}}) = \text{✘}$ hold and $c'_{\text{✘}}$ is a *relevant configuration*—that is, no single circumstance of $c'_{\text{✘}}$ can be removed from $c'_{\text{✘}}$ to make the failure disappear.
- The *ddmin* algorithm is defined as *ddmin*$(c_{\text{✘}}) =$ *ddmin*$'(c'_{\text{✘}}, 2)$ with

$$
\mathit{ddmin}'(c'_{\text{✘}}, n) = \begin{cases} c'_{\text{✘}} & \text{if } |c'_{\text{✘}}| = 1 \\ \mathit{ddmin}'(c'_{\text{✘}} \setminus c_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1 \ldots n\} \times \mathit{test}\ (c'_{\text{✘}} \setminus c_i) = \text{✘} \\ & \quad \text{("some removal fails")} \\ \mathit{ddmin}'(c'_{\text{✘}}, \min(2n, |c'_{\text{✘}}|)) & \text{else if } n < |c'_{\text{✘}}| \text{ ("increase granularity")} \\ c'_{\text{✘}} & \text{otherwise} \end{cases}
$$

where $c'_{\text{✘}} = c_1 \cup c_2 \cup \cdots \cup c_n$ such that $\forall c_i, c_j \times c_i \cap c_j = \emptyset \wedge |c_i| \approx |c_j|$ holds.

The recursion invariant (and thus precondition) for *ddmin*$'$ is *test*$(c'_{\text{✘}}) = \text{✘} \wedge n \leqslant |c'_{\text{✘}}|$.

ddmin is a "minimizing" variant of delta debugging.) Delta debugging again is is an instance of *adaptive testing*—a series of tests in which each test depends on the results of earlier tests.

Let's turn the abstract *ddmin* algorithm into a concrete implementation. Example 5.4 shows a PYTHON implementation of *ddmin*, in which *ddmin*'s tail recursion and existential quantifiers have been changed into nested loops. The implementation relies on a function `split(l, n)`, which splits a list `l` into `n` sublists of roughly equal size (Example 5.5). The function `listminus(c1, c2)` returns a list of all elements that are in the list `c1` but not in the list `c2` (Example 5.6). The constants `PASS`, `FAIL`, and `UNRESOLVED` ✘, and ?, respectively.

The `while` and `if` constructions have the usual meaning (as in C-like languages, the `break` statement leaves the enclosing loop). The `assert` statements document the preconditions and loop invariants.

In addition to these functions, we need an implementation of the `test()` function. Example 5.7 shows a (simplified) version of the `test()` function used by Zeller and Hildebrandt (2002) on a LINUX system. Essentially, it invokes a MOZILLA process and checks its outcome. If it exited normally (indicated by a zero exit status), `test()` returns ✔ (`PASS`). If it crashed (in UNIX: "terminated by a signal 11"), `test()` returns ✘(`FAIL`), and if anything else happens, `test()` returns **?** (`UNRESOLVED`).

EXAMPLE 5.4: A PYTHON implementation of the *ddmin* algorithm

```
def ddmin(circumstances, test):
    """Return a sublist of CIRCUMSTANCES that is a
       relevant configuration with respect to TEST."""

    assert test([]) == PASS
    assert test(circumstances) == FAIL

    n = 2     # Initial granularity

    while len(circumstances) >= 2:
        subsets = split(circumstances, n)
        assert len(subsets) == n

        some_complement_is_failing = 0
        for subset in subsets:
            complement = listminus(circumstances, subset)

            if test(complement) == FAIL:
                circumstances = complement
                n = max(n - 1, 2)
                some_complement_is_failing = 1
                break

        if not some_complement_is_failing:
            if n == len(circumstances):
                break
            n = min(n * 2, len(circumstances))

    return circumstances
```

EXAMPLE 5.5: A PYTHON implementation of the `split()` function

```
def split(circumstances, n):
    """Split a configuration CIRCUMSTANCES into N subsets;
       return the list of subsets"""

    subsets = []   # Result
    start = 0      # Start of next subset
    for i in range(0, n):
        len_subset = int((len(circumstances) - start) /
                         float(n - i) + 0.5)
        subset = circumstances[start:start + len_subset]
        subsets.append(subset)
        start = start + len(subset)

    assert len(subsets) == n
    for s in subsets:
        assert len(s) > 0

    return subsets
```

EXAMPLE 5.6: A PYTHON implementation of the `listminus()` function

```
def listminus(c1, c2):
    """Return all elements of C1 that are not in C2.
       Assumes elements of C1 are hashable."""

    # The hash map S2 has an entry for each element in C2
    s2 = {}
    for delta in c2:
        s2[delta] = 1

    # Check elements in C1 whether they are in S2
    c = []
    for delta in c1:
        if not s2.has_key(delta):
            c.append(delta)

    return c
```

EXAMPLE 5.7: A (simplified) PYTHON implementation of the `test()` function

```
def test(c):
    # Create Mozilla input file
    write_html(c, "input.html")

    parent = os.fork()
    if parent < 0:
        # fork() failed - no more processes
        sys.exit(1)

    elif not parent:
        # Invoke Mozilla
        # TODO: Replay user interaction, too
        os.execv("/usr/bin/mozilla",
                 ["mozilla", "input.html"])

    # Wait for Mozilla to complete
    childpid, status = os.waitpid(parent, 0)
    if os.WIFEXITED(status):
        exit_status = os.WEXITSTATUS(status)
        if exit_status == 0:
            return PASS     # Exited normally

    if os.WIFSIGNALED(status):
        caught_signal = os.WTERMSIG(status)
        if caught_signal == 11:
            # TODO: Check backtrace, too
            return FAIL     # Crashed w/ signal 11

    return UNRESOLVED
```

The full-fledged implementation (not listed here) additionally replays recorded user interaction to trigger the failure (see Chapter 4). It also instruments a debugger (see Chapter 8) to check the *backtrace*—the stack of functions active at the moment of the crash—and returns `FAIL` only if the found backtrace is identical to the backtrace of of the original failure—that is, the program crashes at the same place in the same way.

What happens if we actually run this PYTHON implementation? Example 5.8 shows all tests conducted by delta debugging. The initial tests are run as sketched in Section 5.4—cutting away large chunks of input first, and smaller chunks later. At test 33, *ddmin* has actually reached the relevant input `<SELECT>`. The remaining tests demonstrate that every single character in `<SELECT>` is relevant for the failure to occur.

EXAMPLE 5.8: Simplifying by characters

```
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⟩✔
 2  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨20⟩✔
 3  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨30⟩✔
 4  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨30⟩✗
 5  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨20⟩✔
 6  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨20⟩✗
 7  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨10⟩✔
 8  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨10⟩✔
 9  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨15⟩✔
10  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨15⟩✔
11  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨15⟩✗
12  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨10⟩✔
13  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨10⟩✔
14  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨10⟩✔
15  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨12⟩✔
16  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨13⟩✔
17  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨12⟩✔
18  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨13⟩✗
19  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨10⟩✔
20  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨10⟩✔
21  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨11⟩✔
22  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨10⟩✗
23  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨ 7⟩✔
24  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨ 8⟩✔
25  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨ 7⟩✔
26  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨ 8⟩✔
27  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨ 9⟩✔
28  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨ 9⟩✔
29  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨ 9⟩✔
```

```
30  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨9⟩✔
31  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨8⟩✔
32  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨9⟩✔
33  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨8⟩✘
34  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
35  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
36  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
37  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
38  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
39  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨6⟩✔
40  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
41  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
42  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
43  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
44  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
45  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
46  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
47  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔
48  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>⟨7⟩✔

                      Result: <SELECT>
```

This property—every remaining circumstance in $c'_{\text{✘}} = ddmin(c_{\text{✘}})$ being relevant—is a general property of *ddmin*. Such a configuration is called *relevant configuration* or a *1-minimal configuration* (see Definition A.11 in the Appendix for details). It can be easily proven (see Proposition A.12 in the Appendix) that every configuration returned by *ddmin* is relevant because *ddmin* can return $c'_{\text{✘}}$ only after it tried removing every single element and the failure did not occur for any such configuration.

One should note, though, that $c'_{\text{✘}}$ is not necessarily the *minimal* configuration for which the failure still occurs. To find that out, an algorithm would have to test every subset $c'_{\text{✘}}$—that is, $2^{|c'_{\text{✘}}|}$ tests.

5.6 SIMPLIFYING USER INTERACTION

Delta debugging can be applied to all inputs, as described in Chapter 4. For instance, one can use it to simplify *user input*—leaving only the relevant key strokes and mouse movements in the script. This is especially useful if it is used anyway to reproduce the failure, as in the case of the MOZILLA crash.

To reproduce the failure, Zeller and Hildebrandt (2002) recorded 95 user events, such as pressing keys or buttons and moving moving the mouse. Replaying these 95 events reproduced the they all necessary?

Figure 5.2 shows the progress of *ddmin* applied to these 95 events. The (logarithmic) Y axis shows the number of events $|c'_{\text{✘}}|$ left to be simplified; the (linear) X axis

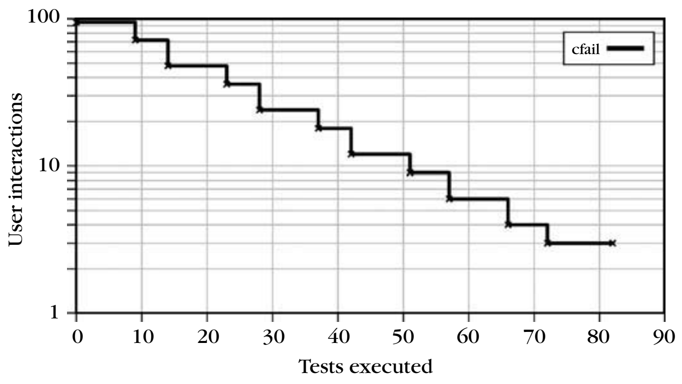

FIGURE 5.2

Simplifying MOZILLA user interactions. After 82 tests, *ddmin* has determined 3 events out of 95 that are required to produce the failure: pressing Alt+P, pressing the mouse button, and releasing it again.

shows the number of tests executed so far. After 82 tests, *ddmin* has simplified the user interaction to only three events:

1. Press the *P* key while the *Alt* modifier key is held. (Invoke the *Print* dialog.)
2. Press the *left mouse button* on the *Print* button without a modifier. (Arm the *Print* button.)
3. Release the *left mouse button*. (Start printing.)

Irrelevant user actions include moving the mouse pointer, selecting the *Print to File* option, altering the default file name, setting the print margins to *.50*, and releasing the *P* key before clicking on *Print* (all of this is irrelevant in producing the failure). (It *is* relevant, though, that the mouse button be pressed before it is released.)

In addition to input in general, delta debugging can be applied to circumstances as they occur during the program run or during the program development. Chapter 13 discusses how to generalize delta debugging to automatically find actual causes in input, code changes, or schedules. Chapter 14 extends this to isolating cause–effect chains within program runs.

5.7 RANDOM INPUT SIMPLIFIED

Another application of automated simplification is to use it in conjunction with *random testing*. The basic idea is to generate large random inputs that trigger a failure and then to simplify the input to reveal the relevant part.

In a classical experiment, Miller et al. (1990) examined the robustness of UNIX utilities and services by sending them *fuzz input*—a large number of random characters. The studies showed that in the worst case 40 percent of the basic programs crashed or went into infinite loops when being fed with fuzz input.

Zeller and Hildebrandt (2002) subjected a number of UNIX utilities to fuzz input of up to a million characters until they until they showed a failure—and then used *ddmin* to failure-inducing input. The first group of programs showed obvious *buffer overrun* problems:

- FLEX (fast lexical analyzer generator), the most robust utility, crashed on sequences of 2,121 or more non–new line and non-NUL characters.
- UL (underlining filter) crashed on sequences of 516 or more printable non–new line characters.
- UNITS (convert quantities) crashed on sequences of 77 or more 8-bit characters.

The second group of programs appeared vulnerable to *random commands*:

- The document formatters NROFF and TROFF crashed on *malformed commands* such as `\D^J%0F` and on *8-bit input* such as *Â* (ASCII code 194).
- CRTPLOT crashed on one-letter inputs `t` and `f`.

All of these simplified test cases can directly be associated to a piece piece of code that handles these inputs—and thus to the question.

5.8 SIMPLIFYING FASTER

As Zeller and Hildebrandt (2002) report, the number of tests required increased with the length of the simplified input. Whereas the NROFF and TROFF tests typically required about 100 to 200 test runs, the FLEX tests required 11,000 to 17,960 test runs. Although a single test run need not take more than a few hundredths of a second, this raises the question on how to reduce the number of test cases and to improve the speed.

As shown in Example 5.8, simplifying a configuration c_{x} with *ddmin* requires at least $|c'_{\text{x}}|$ tests, as every single circumstance in the resulting c'_{x} must be tested once. In the worst case, the number of tests can even be quadratic with respect to $|c_{\text{x}}|$, resulting in a maximum number of tests $t = (|c_{\text{x}}|^2 + 7|c_{\text{x}}|)/2$ (for details, see Proposition A.13 in the Appendix). Although this is quite a pathological example, we should strive to get simplification done as quickly as possible.

5.8.1 Caching

The *ddmin* algorithm does not guarantee that each configuration is tested only once. Thus, our simple `ddmin()` implementation in Example 5.4 may invoke the `test` function multiple times for the same configuration. In Example 5.8, for instance, the six tests runs 41–45 and 48 have been executed before. By using a cache to store test outcomes, one could return the earlier test outcome whenever a test is repeated.

5.8.2 Stop Early

Why does it take so long to simplify the FLEX input? Figure 5.3 shows the first 500 steps of the *ddmin* algorithm. You can easily see that the size quickly decreases, but after about 50 tests progress is very slow (and continues this way for the next 10,500 tests).

Normally, there is no need to try to squeeze the very last character out of an input. One can simply stop the simplification process:

- When a certain granularity has been reached ("We don't care about removing single characters").
- When no progress has been made ("In the last 100 iterations, the size of the input has been reduced by 1 percent only").
- When a certain amount of time has elapsed ("One hour of automatic simplification suffices").

In the case of FLEX, any of these heuristics could stop minimization early.

5.8.3 Syntactic Simplification

One effective way of speeding up simplification is to simplify not by characters but by *larger entities*. As shown in Figure 5.1, simplifying the HTML input by *lines* requires but 12 tests to get down to a single line. And indeed, if only *one* circumstance (i.e., one line) is failure-inducing it can be shown that *ddmin* is as efficient as a binary search (see Proposition A.14 in the Appendix).

This idea of *grouping* circumstances can be taken further by simplifying input not at the *lexical* level (i.e., characters, words, or lines) but at a *syntactical* level—that is, simplifying while preserving the syntactic structure of the input. The basic idea of syntactic simplification is to turn the input into a tree (formally, a *syntax*

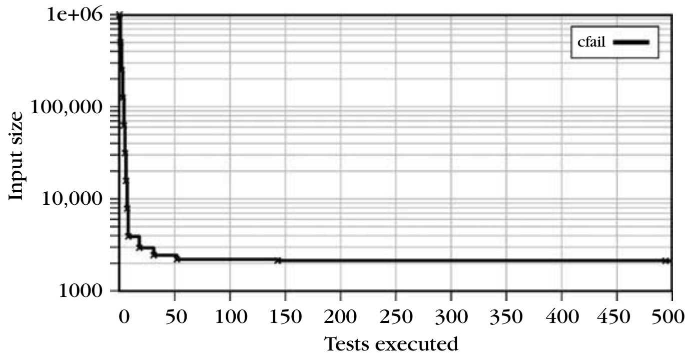

FIGURE 5.3

Simplifying FLEX input. Initially, *ddmin* quickly decreases the size of the input, but then simplifying shows no significant progress.

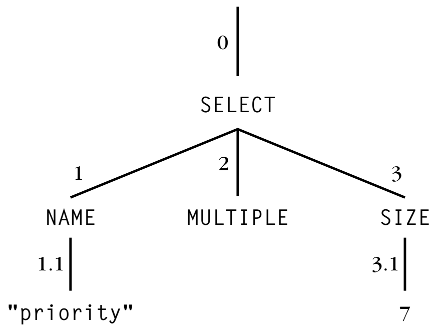

FIGURE 5.4

An HTML tree. Simplifying such a tree structure rather than plain text yields better performance.

tree) and to simplify the tree rather than the input string. An HTML or XML tree representing our MOZILLA one-line example, for instance, would look like that shown in Figure 5.4.

To simplify such a tree, we make every *node* a *circumstance*. Our initial failing configuration thus contains six nodes rather than 40 characters. The *test* function accepting a configuration would remove missing nodes from the HTML tree (rather than cutting away chunks of the input string) and create an HTML input from the remaining tree and feed it to MOZILLA.

However, what would *test* do if asked by *ddmin* to test an *infeasible configuration*? In Figure 5.4, for instance, we cannot remove node 1, `NAME`, without also removing its child node 1.1, "`priority`." Vice versa, HTML rules dictate that the `NAME` attribute must have a value. The "`priority`" node is thus mandatory.

To cope with such syntactic and semantic constraints, the *test* function should be set up to simply return **?** ("unresolved") whenever a configuration is impossible to achieve. That is, *test* does not run MOZILLA, but immediately returns **?** such that *ddmin* selects the next alternative. Furthermore, the splitting of a configuration into subsets can be set up to take constraints into account—for by also keeping nodes that are in the same subtree in the same subset.

Figure 5.5 shows the tests actually carried out within the *ddmin* run. Our initial configuration is $c_{\text{✘}} = \{0, 1, 1.1, 2, 3, 3.1\}$, standing for "all nodes are present."

ddmin tries the configuration {2, 3, 3.1}, which is infeasible. Removing the second half works and gets us $c'_{\text{✘}} = \{0, 1, 1.1\}$. In the next iteration, the configuration {1, 1.1} is infeasible, but {0} is fine. *ddmin* is done, having required just two actual tests to simplify the HTML input.

5.8.4 Isolate Differences, Not Circumstances

Instead of simplifying *all* circumstances, one can also simplify a *difference* between a configuration that works and a configuration that fails. Example 5.9, which illustrates this idea again shows two MOZILLA inputs: one that makes MOZILLA fail and one that

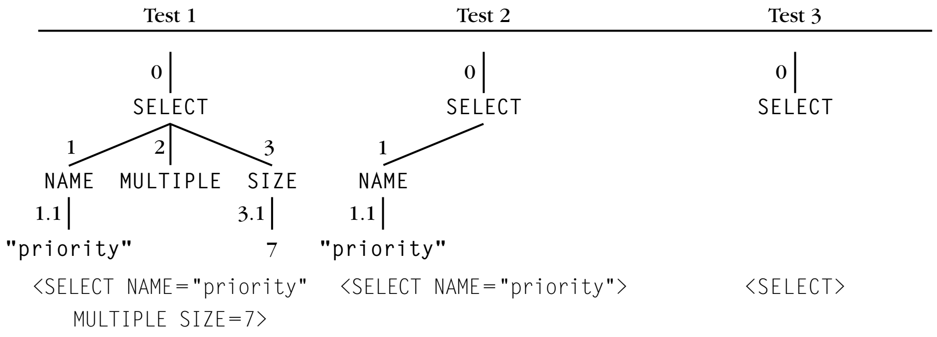

FIGURE 5.5

Simplifying an HTML tree in three steps.

EXAMPLE 5.9: A failure-inducing difference. The initial < sign is isolated as the failure cause

```
<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>   (40 characters)   ✗
<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>   (39 characters)   ✓
```

makes it pass. Neither of these two inputs is simplified. However, their *difference* has been simplified to just one character—the leading < sign. If the < sign is missing, MOZILLA interprets the input as ordinary text instead of as an HTML tag. Thus, whether the < is part of the input or not determines whether MOZILLA fails or not.

Why would one want to simplify differences? There are two answers:

1. *Focusing.* As we will see in Chapter 12, a difference between a configuration that works and a configuration that fails is a *failure cause*—and this failure cause is more precise the smaller the difference is. Thus, a small difference can pinpoint the problem cause in a common context.

   As an example of a common context, think about simplifying user interactions. Minimizing user interaction may still end up in 1,000 interactions or so, all required to set up the context in which the failure occurs. Isolating a difference, though, will reveal a (minimal) difference that decides the final outcome: "If the user had not selected this option, the test would have passed."

2. *Efficiency.* Differences can be simplified faster than entire configurations. This is so because each passing test can be exploited to reduce the difference. Using minimization, only failing tests help in minimizing the configuration. As an example of efficiency, an algorithm that isolates the single < difference requires only 5 tests (compared to the 48 tests in Example 5.8).

More on the isolation of failure causes automatically, as well as a discussion of the involved algorithms and techniques, can be found in Chapter 13.

5.9 CONCEPTS

The aim of simplification is to create a simple *test case* from a detailed problem report (Section 5.1).

Simplified test cases (Section 5.2):

- Are easier to communicate
- Facilitate debugging
- Identify duplicate problem reports

How To

To simplify a test case, remove all irrelevant circumstances. A circumstance is irrelevant if the problem occurs regardless of whether the circumstance is present or not (Section 5.3).

To automate simplification, set up:

- An *automated test* that checks whether the problem occurs
- A *strategy* that determines the relevant circumstances

One such strategy is the *ddmin delta debugging algorithm* (Section 5.4).

Circumstances to be simplified include not only the program input as data, but all circumstances that might affect the program's outcome—for instance, user interactions (Section 5.6).

Simplification can be combined with *random testing* to reveal the failure-inducing parts of the input.

To speed up automatic simplification, employ one or more of the following:

- Make use of caching
- Stop early
- Simplify at a syntactic or semantic level
- Isolate failure-inducing differences rather than circumstances

These techniques are described in Section 5.8.

5.10 TOOLS

Delta Debugging. A full PYTHON implementation of *ddmin* is available at *http://www.st.cs.uni-saarland.de/dd/*.

Simplification Library. Daniel S. Wilkerson of the University of California at Berkeley has another implementation of *ddmin*. This is found at *http://freshmeat.net/projects/delta/*.

5.11 FURTHER READING

Manual simplification of test cases is a long-known programming (or debugging) technique. I recommend Kernighan and Pike (1999) for anyone who wants further depth on the subject. McConnell (1993) also highlights the importance of simplifying a test case as the first step in debugging.

The principle of *divide-and-conquer* is often attributed to the Romans (*divide et impera*) as a governing principle. Its first explicit usage as a political maxime was by the Florentine political philosopher Niccolò Machiavelli (1469–1527), denouncing the motto of Louis XI of France in dealing with his nobles.

As far as I know, the work of Zeller and Hildebrandt (2002) was the first general approach to automatic test case simplification. Details on delta debugging applied to program input are listed. The article also includes a set of case studies, including an in-depth examination of the MOZILLA example. Ralf Hildebrandt and I had much fun conducting this research. I hope you'll have some fun reading the article.

Note that the *ddmin* algorithm as described in Zeller and Hildebrandt (2002) slightly differs from the version presented here. The "old" *ddmin* algorithm had an additional check whether one of the subsets c_i would fail—that is, $test(c_i) = ✗$ holds—and if so would reduce $c'_✗$ to c_i. This extra test has shown few benefits in practice, which is why it is not included here.

The Gecko BugAThon is still going on, and you can still contribute—automatically or manually. At the time of this writing, you would get a stuffed Firefox for every 15 bugs simplified. For details, have a look at *https://developer.mozilla.org/en/Gecko_BugAThon*.

EXERCISES

For some of the following exercises you need PYTHON or JAVA:

- Using PYTHON, you can start immediately using the code examples in this chapter. PYTHON interpreters are available at *http://www.python.org/*.
- Using JAVA, you can use the JAVA class in Example 5.10 as a starting point. All it needs are `split()` and `listminus()` functions as those defined for PYTHON. The `test()` function is designed to be overloaded in a problem-specific subclass.
- If you know neither PYTHON nor JAVA, adapt the code to a language of your choice.

5.1 The function `bool geegg(string s)` returns

- `true` if the string `s` contains three `g` characters or more, or
- `true` if `s` contains two `e` characters or more, and
- `false` otherwise.

For instance, `geegg("good eggs tomorrow")` returns `true`, `geegg("no eggs today")` returns `false`.

Apply the *ddmin* algorithm on the 16-character input

```
a-debugging-exam
```

EXAMPLE 5.10: A delta debugging class in JAVA

```
import java.util.LinkedList;
import java.util.List;
import java.util.Iterator;

public class DD {
    // Outcome
    public static final int FAIL       = -1;
    public static final int PASS       = +1;
    public static final int UNRESOLVED = 0;

    // Return a - b
    public static List minus(List a, List b) { ... }

    // test function - to be overloaded in subclasses
    public int test(List config) { return UNRESOLVED; }

    // Split C into N sublists
    public static List split(List c, int n) { ... }

    // ddmin algorithm
    // Return a sublist of CIRCUMSTANCES that is a relevant
    // configuration with respect to TEST.
    public List ddmin(List circumstances_) {
        List circumstances = circumstances_;

        assert test(new LinkedList()) == PASS;
        assert test(circumstances) == FAIL;

        int n = 2;

        while (circumstances.size() >= 2) {
            List subsets = split(circumstances, n);
            assert subsets.size() == n;

            boolean some_complement_is_failing = false;
            for (int i = 0; i < subsets.size(); i++) {
                List subset = (List)subsets.get(i);
                List complement = minus(circumstances, subset);

                if (test(complement) == FAIL) {
                    circumstances = complement;
                    n = Math.max(n - 1, 2);
                    some_complement_is_failing = true;
                    break;
                }
            }

            if (!some_complement_is_failing) {
                if (n == circumstances.size())
                    break;
                n = Math.min(n * 2, circumstances.size());
            }
        }

        return circumstances;
    }
}
```

to find a 1-minimal input that still causes `geegg()` to return true. Record the individual inputs and test outcomes.

5.2 As recent versions of MOZILLA tend to be much more stable, we shall simulate the "old" MOZILLA behavior using a `test()` function (Example 5.10).

(a) Implement a PYTHON or JAVA `test()` function that accepts a list of characters. If the list contains the string `<SELECT>`, have it return `FAIL`, and `PASS` otherwise.

(b) The version of `test()` in question (a) does not take extra attributes of `SELECT` into account. Have it return `FAIL` if the character list contains a substring that matches the *regular expression* `<SELECT *[^>]*>` (that is, `<SELECT` followed by zero or more blank characters, followed by zero or more characters other than `>`, and finally followed by `>`).

5.3 Use your `test()` function to minimize the problem report in Figure 3.1 manually and systematically. How many executions of `test()` do you need?

5.4 Using `test()`, `split()`, and `listminus()`, use your implementation of *ddmin* to simplify the problem report in Figure 3.1 automatically. If you simplify by lines, how many tests do you need?

5.5 Repeat Exercise 5.4, simplifying the remaining line by characters.

5.6 Repeat Exercise 5.4, simplifying syntactically. Use an XML parser to read in the HTML input. Use syntactic simplification as sketched in Section 5.8.3.

5.7 Design an input and a `test()` function such that *ddmin* requires a maximum of test runs. (Hint: See Zeller and Hildebrandt (2002) for the discussion of worst-case behavior.)

5.8 Design a `split()` function for plain text that attempts to keep paragraphs, lines, and words together as long as possible.

5.9 The *ddmin* algorithm only finds one possible 1-minimal input (i.e., an input where removing any single character makes the failure disappear). Sketch an extension of `ddmin()` that finds *all* possible 1-minimal inputs. Sketch its complexity.

5.10 Isolating a minimal failure-inducing difference using only a `test()` function has exponential complexity. Prove this claim.

Perfection is achieved not when you have nothing more to add, but when there is nothing left to take away.

– (attributed to) ANTOINE DE SAINT-EXUPÉRY