

# From Automated Testing to Automated Debugging

Andreas Zeller  
Lehrstuhl für Software-Systeme  
Universität Passau, Germany  
*zeller@computer.org*

## Abstract

Debugging is still one of the hardest, yet least systematic activities of software engineering. The Delta Debugging algorithm isolates failure causes automatically—by systematically narrowing down failure-inducing circumstances until a minimal set remains. Delta Debugging has been applied to isolate failure-inducing program input (e.g. a HTML page that makes a Web browser fail), failure-inducing user interaction (e.g. the keystrokes that make a program crash), or failure-inducing changes to the program code (e.g. after a failing regression test).

Delta Debugging is fully automatic; all it requires is an automated test that detects whether the expected failure is present or not. The method is introduced using real-life programs with real-life bugs.

## Introduction

In the last 50 years, we have seen tremendous productivity increases in all areas of software development. Except for one area. Debugging computer programs—the process of identifying and correcting the root cause of a failure—is still as labor-intensive and painful as it was 50 years ago. Granted, we have sophisticated debugging tools that help examining program states during execution. But still, debugging is more art than engineering—dependent on the skills and the experience of the specific programmer.

Debugging is so hard and so neglected because the debugging process is different from all other processes in software engineering. Debugging is experimental science: noticing something, one wonders why it happens, and one sets up a number of hypotheses which one confirms or refutes by means of experiments. This “trial-and-error” process can be very chaotic and intuition-driven. But there is no reason why debugging should not be conducted as disciplined, systematic, and quantifiable as other areas of software engineering—and finally automated, too.

In this paper, we present *Delta Debugging*, a new approach to automated debugging that satisfies all these criteria. The Delta Debugging algorithm automatically isolates *failure-inducing circumstances*—that is, those factors that are necessary for producing a specific program failure (Figure 1 on the following page). Delta Debugging is *disciplined*, because every circumstance is taken into account. Delta Debugging is *systematic*, because it automatically and systematically narrows down the set of possible circumstances. Finally, Delta Debugging is *quantifiable*, because it has been applied successfully for a number of failure-inducing circumstances. These circumstances include:

**Program input** Delta Debugging automatically isolates failure-inducing input.

**User interaction** Delta Debugging automatically simplifies failure-inducing user interaction.

**Changes to the program code** Delta Debugging automatically identifies failure-inducing code changes, e.g. in regression testing.

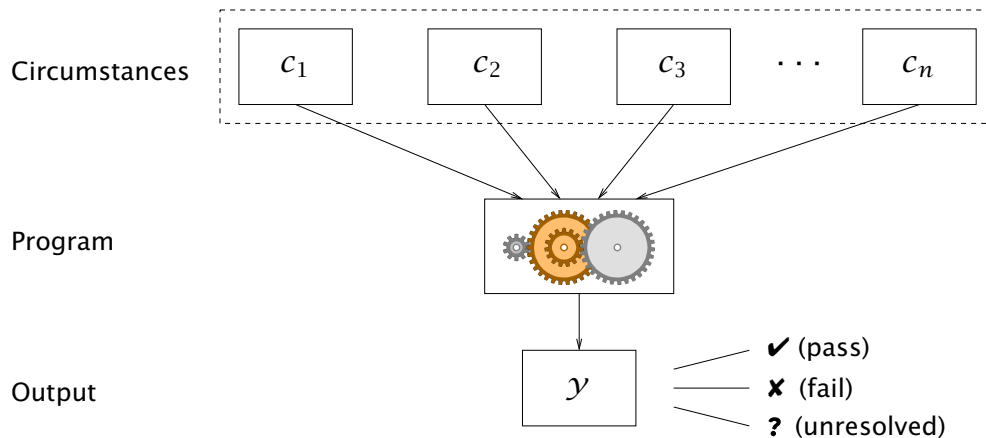


Figure 1: **How circumstances affect program behavior.** From the debugger’s view, a program outcome is determined by several circumstances—the program’s input, its code, its environment, and so on. The goal is to isolate the circumstances that are relevant for the program outcome from those which are irrelevant.

Delta Debugging is fully automatic: You specify the program to be executed as well as an automated test function that tells whether a program run has shown the expected failure or not. Delta Debugging then runs the program over and over again under changed circumstances, in order to separate the failure-inducing circumstances from the irrelevant ones. This also means that Delta Debugging has a cost: the number of tests can be quite large. Nonetheless, Delta Debugging always results in a set of relevant failure-inducing circumstances—circumstances that provide significant insights about the nature and the cause of the failure.

In the following sections, we give an introduction into Delta Debugging using some real-life programs with real-life bugs. We start with a popular Web browser.

## Your Task: “Simplify 370 Bug Reports”

If you browse the Web with Netscape 6, you actually use a variant of *Mozilla*, Netscape’s open source web browser project [2]. As a work in progress with big exposure, the Mozilla project receives several dozens of bug reports a day. The first step in processing any bug report is *simplification*, or to eliminate all details that are irrelevant for producing the failure. Such a simplified bug report not only facilitates debugging, but it also subsumes several other bug reports that only differ in irrelevant details.

In July 1999, *Bugzilla*, the Mozilla bug database, listed more than 370 open bug reports—bug reports that were not even simplified. With this queue growing further, the Mozilla engineers “faced imminent doom” [3]. Overwhelmed with work, Eric Krock, the Netscape product manager, sent out the *Mozilla BugAthon call for volunteers* [3] that would help process bug reports: For 5 bug reports simplified, a volunteer would be rewarded with an invitation to the launch party; 20 bugs would earn him a T-shirt signed by the grateful engineers. “Simplifying” meant: turning these bug reports into *minimal test cases*, where every part of the input would be significant in reproducing the failure.

Here is an actual example: Bugzilla entry #24735, reported by *anantk@yahoo.com*.

```
Ok the following operations cause mozilla to crash consistently on my ma-
chine
→ Start mozilla
```

- Go to `bugzilla.mozilla.org`
- Select search for bug
- Print to file setting the bottom and right margins to `.50` (I use the file `/var/tmp/netscape.ps`)
- Once it's done printing do the exact same thing again on the same file (`/var/tmp/netscape.ps`)
- This causes the browser to crash with a `segfault`

In order to simplify this bug report, a volunteer is supposed to load the Bugzilla Web page (<http://bugzilla.mozilla.org/>) into his text editor. Then, as the BugATHon instructions state,

Start removing HTML markup, CSS rules, and lines of JavaScript from the page. Start by removing the parts of the page that seem unrelated to the bug. Every few minutes, check the page to make sure it still reproduces the bug. [...] When you've cut away as much HTML, CSS, and JavaScript as you can, and cutting away any more causes the bug to disappear, you're done. [3]

This method is an instance of Occam's Razor: Everything that is not strictly required for explaining a phenomenon can be cut away. In experimental science as in debugging, each cut is verified by an experiment—in our case, by repeating the Mozilla run with the simplified HTML page. But there is no reason why this process should be carried out manually—especially in an environment designed for automation. If we have an automated test that tells us whether the failure is still present or not, we can easily *automate the entire simplification process*.

## Separating the Wheat from the Chaff

Setting up an automated test that checks whether printing a specific HTML page works is not very difficult. All one needs is a *record/replay* facility that will record user interaction in order to replay it later. With such a tool, one can easily verify automatically whether printing a specific HTML page makes Mozilla crash.

To automate the simplification process, one could write a program that removes single characters from the input; after every removal, it would check (using the replay tool) whether the failure still occurs. This is repeated until every single remaining character is relevant for producing the failure.

Unfortunately, this approach is ineffective: For the, say, 40,000 characters of the Bugzilla web page, we need at least 40,000 tests. And since it may be that we only succeed in cutting away the last character again and again, after having tried to remove each of the other characters, the number of tests may be even greater—for  $n$  characters, up to  $n^2/2$  tests.

A more effective approach is to use the same technique as experienced programmers: Start by cutting away large chunks first and increase granularity later. For instance, we may start by cutting away chunks of 20,000 characters, then chunks of 10,000, 5,000, 2,500 characters, and so on, cutting away anything that is not relevant for the failure. This granularity can be increased until, eventually, we end up removing single characters.

This is what we did. We built a prototype called WYNOT (for “worked yesterday, not today”) that implemented this approach. WYNOT takes an input and runs a test on it (such as replaying a Mozilla user interaction). Starting with large chunks, WYNOT keeps on removing parts of the input until the chunk size is minimal and the input can no longer be simplified.

In a case study, we reproduced the “Mozilla cannot print” failure as reported above and ran WYNOT using the Bugzilla HTML code as input for Mozilla. After 57 test runs (each starting Mozilla and replaying the previously recorded user interaction), WYNOT simplified the original 896 lines to a 1-line input:

```
<SELECT_NAME="priority" _MULTIPLE_SIZE=7>
```

```

1 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✗
2 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
3 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
4 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
5 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✗
6 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✗
7 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
8 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
9 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
10 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✗
11 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
12 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
13 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
14 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
15 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
16 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✗
17 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✗
18 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✗
19 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
20 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
21 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
22 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
23 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
24 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
25 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
26 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✗

```

Figure 2: **Simplifying HTML input.** In the last stages of simplification, the WYNOT prototype cuts away smaller and smaller chunks of failure-inducing input until the input can no longer be simplified—only a `<SELECT>` tag remains. The test outcome is indicated by ✓ (pass) and ✗ (failure); gray characters have been cut away from the input.

This single line was then reduced further in another 25 runs. Figure 2 shows how WYNOT systematically tests Mozilla while removing smaller and smaller chunks from the input. Whenever the test fails (✗), the chunk just removed stays removed in all later tests. Eventually, only the `<SELECT>` tag remains. With an average of 15 seconds per test run (on a 400 MHz Linux PC) and a total of 82 tests, the whole process took 21 minutes.

## The Keystroke that made Mozilla Crash

So, we have now simplified the HTML page, but what about the other details of the bug report? Is “setting the bottom and right margins to .50” really necessary? After all, user actions are but another kind of program input, and can be simplified likewise. We thus subjected the log of recorded user interactions to WYNOT, in order to find a *failure-inducing minimum of user actions*. The 95 user actions included moving the mouse pointer, pressing or releasing a mouse button, and pressing or releasing a key on the keyboard.

After 82 test runs (or 21 minutes), only 3 out of these 95 user actions were left:

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

The removed user actions included 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 these were irrelevant in producing the failure. (It was relevant, though, that the mouse button be pressed before it was released.)

What we ended up after simplifying both HTML input and user actions was a simplified bug report:

Printing a page containing `<SELECT>` makes Mozilla crash.

That’s all—everything else was irrelevant. As long as the bug reports can be reproduced, this minimization procedure could easily be repeated automatically with the 10,700 other bugs (as of November 10, 2000) in the Bugzilla database. All one needs is a HTML input, a sequence of user actions, an observable failure—and a little time to let the computer simplify the failure-inducing input.

Besides HTML input and user actions, WYNOT has been tested on 44 more input-related failures, always simplifying the input significantly [1].

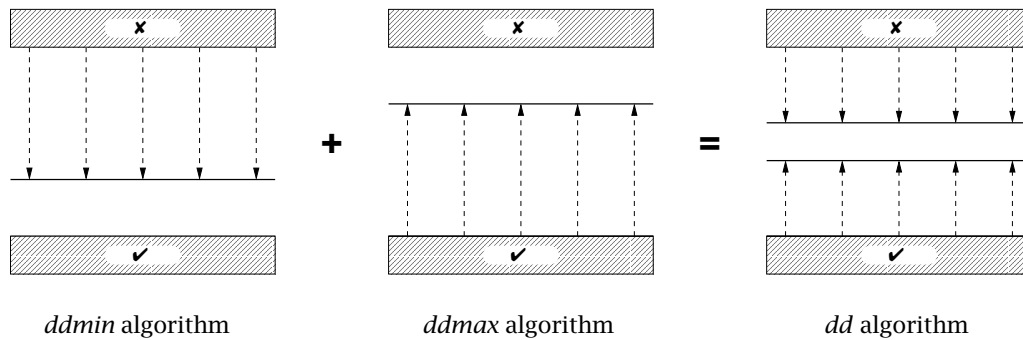


Figure 3: **Narrowing strategies.** To isolate failure-inducing differences between a working run (bottom) and a failing run (top), one can remove differences from the failing run (left). But one can also add differences (center) or narrow down the set from both sides (right). In all three cases, the remaining difference contains the failure-inducing circumstances.

## Finding Failure-Inducing Code

The input of a program is only one of many circumstances that affect its behavior. There is the program environment, the operating system, the runtime library...—anything that can affect the outcome may be a relevant circumstance. Currently, we are exploring Delta Debugging for circumstances like

- *environment settings* (which part of the environment is relevant?),
- *system libraries* (after an upgrade, I am stuck in DLL hell—why?), or
- *thread schedules* in parallel programs (which part of the schedule causes a non-deterministic behavior?).

However, the most important circumstance is, of course, the *program code* itself. So far, we have seen Delta Debugging being applied to program input. In theory, though, we view the program code as input as well—as input to some machine executing the program. Couldn't we apply Delta Debugging to this program code input as well?

In theory, yes—just cut away program code until only a relevant slice remains. In practice, however, we would face a major problem: The code of any nontrivial program is far more *interwoven* than typical program inputs. Most changes such as cutting away statements (or functions, or whatever decomposition) will lead to unresolved test outcomes. This gets us closer to the worst case—i.e., a quadratic number of tests.

To reduce the problem size, additional *program analysis* comes in handy; especially program slicing [4] can eliminate several irrelevant program parts right from the start. The integration of program analysis and Delta Debugging, however, has not yet been explored sufficiently.

But there is another approach that is even more promising: If we know that there is another version of the program where the failure does not show up, we can focus on the *code differences* rather than on the entire code. In other words: Whenever a regression test shows that something used to work, but works no more, we can use Delta Debugging to isolate the failure-inducing difference.

## The Difference that Causes a Failure

To narrow down the difference between two settings, one basically has the choice between three approaches, illustrated in Figure 3. Let us assume we have a working (✓) and a failing

```

2 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✘
4 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✘
  ⋄
7 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
6 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
5 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
3 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓
1 <SELECT_NAME="priority" _MULTIPLE_SIZE=7> ✓

```

Figure 4: **Focusing on the difference.** Narrowing down the difference from both the passing run (✓, bottom) and the failing run (✘, top) is much more efficient than simplifying. After only seven tests, the failure-inducing difference is isolated: the left angle of the HTML tag.

(✘) run. First, we can remove differences from the failing run. If the passing run is the empty set, this is exactly the simplification process as described earlier: we cut away differences until a minimal set remains.

However, there is a better strategy. Rather than only cutting away while the failure persists, one can also *add differences* while the program still passes the test. To get the best efficiency, one can combine both approaches and narrow down the set of differences whenever a test either passes or fails.

In Figure 4, we apply this combined approach to the Mozilla example described earlier. Again, we start with a 1-line input containing the SELECT line—this is the input for the failing run. The working run has an empty input.

After only seven tests, the failure-inducing difference is narrowed down to one character: The input of the failing run is reduced to

```
<SELECT_NAty" _MULTIPLE_SIZE=7>
```

while the input of the working run has been expanded to:

```
SELECT_NAty" _MULTIPLE_SIZE=7>
```

The difference is only the first character, changing the HTML tag to ordinary text. So, again, the failure only depends on whether a SELECT tag is printed or whether ordinary HTML text is printed.

While narrowing down the difference is in general more efficient than a one-sided approach, there is an important problem: The test outcome may be neither “pass” nor “fail”, but *unresolved*—that is, the given set of circumstances is inconsistent. This problem frequently occurs when dealing with changes to program code, as shown in the next example.

## After Changing 178,000 Lines of Code, GDB no Longer Works

Let us now come back to finding failure-inducing differences in program code. GNU DDD is a graphical front-end for UNIX command-line debuggers such as the GNU debugger (GDB). In July 1998, Brian Kahne from Motorola sent in a bug report stating that after upgrading GDB from version 4.16 to 4.17, running the debuggee from DDD no longer worked properly:

```

Date: Fri, 31 Jul 1998 15:11:05 -0500
From: Brian Kahne <bkahne@ibmoto.com>
To: DDD Bug Report Address <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. [...]

```

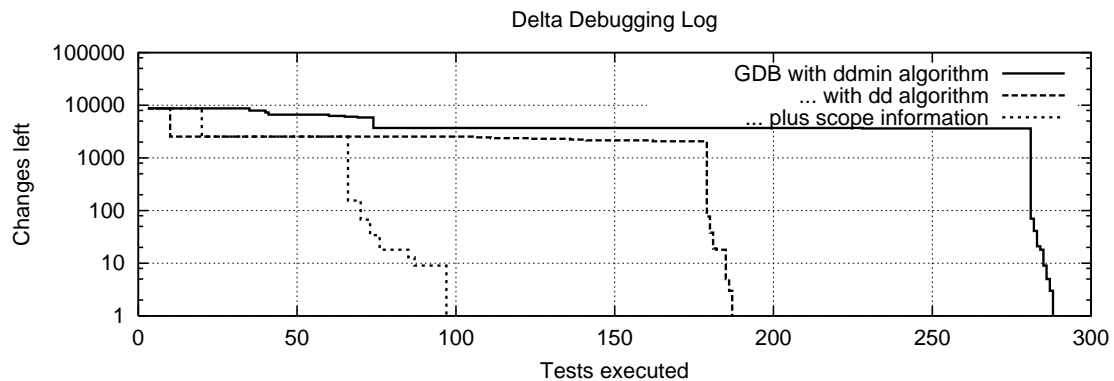


Figure 5: **Delta Debugging optimizations.** Minimizing the set of changes between GDB 4.16 and GDB 4.17 isolates the failure-inducing change after 288 tests. Narrowing down the difference between the two versions finds the same change after 187 tests. Grouping changes according to scope reduces the number of tests down to 97.

This is a classical instance of the “worked yesterday, not today” problem: Some change (or difference) occurs, and this change causes a failure. The sole problem in this case is that the change is *huge*: The DIFF between the source codes of GDB 4.16 and GDB 4.17 has a length of 178,000 lines—that is, 178,000 lines have either been added or deleted or changed between the two releases. (If you consider that the GDB source code itself has about 500,000 lines, and that about a third of the code has been changed, you might wonder why everything else still works.) Somewhere within these 178,000 lines lie the changes that caused DDD to fail—but where?

We used Delta Debugging to resolve this problem. First, we decomposed the 178,000 line-DIFF into 8,721 textual changes in the GDB source, with any two textual changes separated by a context of at least two unchanged lines. Now, WYNOT would isolate the failure-inducing changes. Each test applied a subset of changes to the GDB code, rebuilt GDB and ran an automated test which would verify whether the failure was still present.

In this setting, we used the narrowing approach, thus reducing the differences between the two original versions. However, we made an important optimization to our algorithm. Since WYNOT combined changes more or less arbitrarily, most tests turned out unresolved, because the changes would not integrate, the program would not build, and so on. So, rather than grouping changes arbitrarily, we grouped changes by their *scope*, that is, the directory, file, or function in which they were applied. The assumption was that a set of changes was less likely to be in conflict if they all applied to the same scope.

The result of these Delta Debugging runs (and the effect of these optimizations) is shown in Figure 5. Starting with 8,721 changes, WYNOT required between 288 and 97 tests—the more optimizations applied, the less tests were required. On a 400 MHz Linux PC, each test took about 240 seconds to apply the changes, rebuild and run GDB; 97 tests thus required about 6,5 hours.

In all three cases, WYNOT broke down the 178,000 lines down to the same one-line change line that, being applied, caused DDD to malfunction:

```
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. Given the command `show args`, GDB 4.16 gives a

reply which is obviously constructed from the changed string constant:

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

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

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

which could not be parsed by DDD! To solve the problem here and now, we simply reversed the GDB change; eventually, DDD was upgraded to make it work with the new GDB version.

Applying Delta Debugging to find failure-inducing changes has been applied to other projects as well. However, in most of these projects, we had a *version repository* available which would allow us to group changes by their creation date. With such knowledge, the risk of inconsistencies decreases dramatically. In [5], we show how Delta Debugging reduced 344 ordered changes from the DDD version repository to a single failure-inducing one—in only 12 tests.

## Conclusion

We have shown how Delta Debugging isolates failure-inducing program input and failure-inducing code changes for real-life programs with real-life bugs. We expect that Delta Debugging can be applied to arbitrary circumstances of the program execution. All that is required is an automated test which runs the debuggee under chosen circumstances and checks whether a given failure is still present. Delta Debugging thus automates the most time-consuming part of debugging—the localization of failure circumstances.

Implementing Delta Debugging is rather straight-forward. The full Delta Debugging algorithm is shown in Figure 6 on the next page; an implementation in your favorite imperative programming language takes about 100–200 lines of code. Be aware, though, that Delta Debugging requires a lot of automation: Automated reconstruction is nice to have, but automated testing is a must—and the faster the better, due to the large number of tests. Finally, the better the knowledge about the structure of the circumstances, the less tests will be required.

Despite the straight-forward approach, do not expect that all debugging can be automated. A failure is typically not the result of a single cause. Rather, the initial causes, as determined by Delta Debugging, trigger a *causality chain* of effects and causes, which eventually results in the failure. To fix the error, the programmer must still identify the remainder of the causality chain and decide where to break it. Simply breaking the causality chain for a specific failure is easy, but breaking it such that as many failures as possible are eliminated is still a challenging task.

We expect that future combinations of automated testing and program analysis will isolate the remainder of the causality chain as well, and maybe also assist in breaking it properly. But already today, we can have the computer narrow down failure-inducing circumstances automatically. And since computers were built to relieve humans from boring, monotonous tasks—let’s have them do the debugging!

## Acknowledgments

Steven Barlow, Holger Cleve, Kerstin Reese, Torsten Robschink, Gregor Snelting, and Falk Schreiber provided helpful comments on earlier versions of this article.

## References

- [1] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 135–145, Portland, Oregon, August 2000.

- [2] Mozilla web site. <http://www.mozilla.org/>.
- [3] Mozilla web site: The Gecko BugAThon. <http://www.mozilla.org/newlayout/bugathon.html>.
- [4] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.
- [5] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In Oscar Nierstrasz and Michel Lemoine, editors, *Proc. ESEC/FSE'99 - 7th European Software Engineering Conference / 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267, Toulouse, France, September 1999. Springer-Verlag.

**Andreas Zeller** is an assistant professor at Passau University, Germany. His research interests include dynamic program analysis, configuration management, and program visualization. Zeller received a PhD in computer science from Braunschweig University, Germany. Contact him at [zeller@computer.org](mailto:zeller@computer.org).

More information about Delta Debugging is available at <http://www.fmi.uni-passau.de/st/dd/>.

Let  $C$  be the set of all possible circumstances. Let  $test : 2^C \rightarrow \{\mathbf{X}, \checkmark, ?\}$  be a testing function that determines for a test case  $c \in C$  whether some given failure occurs ( $\mathbf{X}$ ) or not ( $\checkmark$ ) or whether the test is unresolved ( $?$ ).

Now, let  $c$  and  $c'$  be test cases with  $c \subseteq c' \subseteq C$  such that  $test(c) = \checkmark \wedge test(c') = \mathbf{X}$ .  $c$  is the “working” test case and  $c'$  is the “failing” test case.

The *delta debugging algorithm*  $dd(c, c')$  narrows down the difference between  $c$  and  $c'$ . It determines a pair  $(d, d') = dd(c, c')$  such that  $c \subseteq d \subseteq d' \subseteq c'$ ,  $test(d) = \checkmark$ , and  $test(d') = \mathbf{X}$  hold and  $|d'| - |d|$  is *1-minimal*—that is, no single circumstance of  $d'$  can be removed from  $d'$  to make the failure disappear or added to  $d$  to make the failure occur.

The  $dd$  algorithm is defined as  $dd(c, c') = dd_2(c, c', 2)$  with

$$dd_2(c, c', n) = \begin{cases} dd_2(c, c \cup c_i, 2) & \text{if } \exists c_i \cdot test(c \cup c_i) = \mathbf{X} \\ dd_2(c' - c_i, c', 2) & \text{else if } \exists c_i \cdot test(c' - c_i) = \checkmark \\ dd_2(c \cup c_i, c', \max(n-1, 2)) & \text{else if } \exists c_i \cdot test(c \cup c_i) = \checkmark \\ dd_2(c, c' - c_i, \max(n-1, 2)) & \text{else if } \exists c_i \cdot test(c' - c_i) = \mathbf{X} \\ dd_2(c, c', \min(2n, |c'| - |c|)) & \text{else if } n < (|c'| - |c|) \\ (c, c') & \text{otherwise} \end{cases}$$

where  $c' - c = c_1 \cup c_2 \cup \dots \cup c_n$  with  $c_i$  pairwise disjoint and  $\forall c_i (|c_i| \approx (|c'| - |c|)/n)$ .

The recursion invariant for  $dd_2$  is  $test(c) = \checkmark \wedge test(c') = \mathbf{X} \wedge n \leq |c'| - |c|$ .

To obtain the *admin* variant [1], let  $test : 2^C \rightarrow \{\mathbf{X}, ?\}$  return  $?$  even if the failure does not occur; for *ddmax*, let  $test : 2^C \rightarrow \{\checkmark, ?\}$  return  $?$  even if the failure occurs.

The early  $dd^+$  variant [5] has worse performance than  $dd$  and should no longer be used.

Figure 6: **The Delta Debugging algorithm in a nutshell.** The function  $dd$  narrows down the difference between two test cases  $c$  and  $c'$ ; its variant *admin* minimizes a test case  $c'$ . An implementation in an imperative programming language takes about 100–200 lines of code.