Simplifying Failure-Inducing Input

Ralf Hildebrandt
Universität Passau
Lehrstuhl Software-Systeme
Innstraße 33
94032 Passau, Germany
ralf.hildebrandt@gmx.de

Andreas Zeller
Universität Passau
Lehrstuhl Software-Systeme
Innstraße 33
94032 Passau, Germany
zeller@acm.org

ABSTRACT

Given some test case, a program fails. Which part of the test case is responsible for the particular failure? We show how our *delta debugging* algorithm generalizes and simplifies some failing input to a *minimal test case* that produces the failure.

In a case study, the Mozilla web browser crashed after 95 user actions. Our prototype implementation automatically simplified the input to 3 relevant user actions. Likewise, it simplified 896 lines of HTML to the single line that caused the failure. The case study required 139 automated test runs, or 35 minutes on a 500 MHz PC.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, diagnostics, testing tools, tracing*

General Terms

Automated debugging, combinatorial testing

1. INTRODUCTION

Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

— Richard Stallman, *Using and Porting* GNU CC

The Mozilla engineers faced imminent doom. In July 1999, more than 370 open bug reports were stored in the bug data base, ready to be simplified. “Simplifying” meant: turning these bug reports into *minimal test cases*, where every part of the input would be significant in reproducing the failure. Overwhelmed with work, the engineers sent out the *Mozilla BugAThon call for volunteers* that would help them 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 [9].

Decomposing specific bug reports into simple test cases does not only trouble the engineers of Mozilla, Netscape’s open source web browser project [8]. The problem arises from generally conflicting issues: A *bug report* must be as specific as possible, such that the engineer can recreate the context in which the program failed. On the other hand, a *test case* must be as simple as possible, because a minimal test case implies a most general context. Thus, a minimal test case not only allows for short problem descriptions and valuable problem insights, but it also subsumes several current and future bug reports.

The striking thing about test case simplification is that no one so far has thought to *automate* this task. Several textbooks and guides about debugging are available that tell how to use binary search in order to isolate the problem—based on the assumption that the test is carried out manually, too. With an *automated* test, however, we can also *automate test case simplification.*

This is what we describe in this paper. Our *delta debugging* algorithm *ddmin* is fed with a test case, which it simplifies by successive testing. *ddmin* stops when a *minimal test case* is reached, where removing any single input entity would cause the failure to disappear. In general, *ddmin* requires a time of $O(n^2)$ given an input of n entities. A well-structured input leads to better performance: in the best case, where a single input entity causes the failure, *ddmin* requires logarithmic time to find the entity. *ddmin* can be tailored with language-specific knowledge.

We begin with a discussion of the problem and the basic *ddmin* algorithm. Using a number of real-life failures, we show how the *ddmin* algorithm detects failure-inducing input and how this test case is isolated and simplified. We close with discussions of related and future work.

2. CONFIGURATIONS AND TESTS

Ian Hickson stayed up until 5:40 a.m. and simplified 18 bugs the first night of the BugAThon.

— *Mozilla BugAThon call*

Let us begin with some basic definitions. First of all, what does a “minimal” test case mean?

For every program, there is some *smallest possible input* that induces a well-defined behavior which does not qualify as a failure. Typically, this is the *empty input,* or something very close. Here are some examples:

- A C compiler accepts an empty translation unit (= an empty C file) as smallest possible input.
- When given an empty input, a WWW browser is supposed to produce a defined error message.
- When given an empty input file, the LaTeX typesetting system is supposed to produce an error message.

It should be noted that the smallest *possible* input is not necessarily the smallest *valid* input; even an invalid input is possible as long as the program does not fail.

Let us now view a *failure-inducing input* C as the result of applying a number of *changes* $\Delta_1, \Delta_2, \ldots, \Delta_n$ to the minimal possible input. This way, we have a gradual *transition* from the minimal possible input (= no changes applied) to C (= all changes applied).

We deliberately do not give a formal definition of a change here. In general, a Δ_i can stand for *any change in the circumstances that influences the execution of the program.* In our previous work, for instance, we had modeled Δ_i as changes to the program code [15]. In this paper, we search for failure-inducing circumstances in the program input; hence, a change is any operation that is applied on the input. The only important thing is that applying all changes results in the failure-inducing set C.

In the case studies presented in this paper, we have always chosen changes as a *lexical decomposition* of the failure-inducing input. That is, each Δ_i stands for a lexical entity that can be present (the change is applied) or not (the change is not applied). As an example, consider a minimal possible input which is empty, and a failure-inducing input consisting of n lines of text. Each change Δ_i would add the i-th line to the empty input, such that applying all changes results in the full set of lines. Modeling changes as lexical decomposition is the easiest approach, but the model can easily extend to other notions of changes.

Still treating changes as given entities, let us now formally define tests and test cases. We can describe any test case between the minimal possible input and C as a *configuration of changes*:

Definition 1 (Test case) *Let $C = \{\Delta_1, \Delta_2, \ldots, \Delta_n\}$ be a set of changes Δ_i. A change set $c \subseteq C$ is called a* test case.[1]

A test case is constructed by applying changes to the minimal possible input:

Definition 2 (Minimal possible input) *An empty test case $c = \emptyset$ is called the* minimal possible input.

We do not impose any constraints on how changes may be combined; in particular, we do not assume that changes are ordered. In the worst case, there are 2^n possible test cases for n changes.

To determine whether a test case induces a failure, we assume a *testing function.* According to the POSIX 1003.3 standard for testing frameworks [5], we distinguish three outcomes:

[1]The definitions in this section are adapted from our previous work [15]. See Section 8 for a discussion.

- The test *succeeds* (PASS, written here as ✔)
- The test has *produced the failure* it was intended to capture (FAIL, written here as ✘)
- The test produced *indeterminate results* (UNRESOLVED, written here as **?**).[2]

Definition 3 (Test) *The function $test : 2^C \to \{✘, ✔, ?\}$ determines for a test case $c \in C$ whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (**?**).*

In practice, *test* would construct the test case by applying the given changes to the minimal possible input, feed the test case to a program and return the outcome.

Let us now model our initial scenario. We have some *minimal possible input* that works fine and some test case that fails:

Axiom 4 (Failing test case) *The following holds:*

- *$test(\emptyset) =$ ✔ ("minimal input") and*
- *$test(C) =$ ✘ ("failing test case").*

Our goal is now to simplify the failing test case C—that is, to minimize it. A test case c being "minimal" means that no subset of c causes the test to fail. Formally:

Definition 5 (Minimal test case) *A test case $c \subseteq C$ is* minimal *if*

$$\forall c' \subset c \left(test(c') \neq ✘\right)$$

holds.

This is what we want: minimizing a test case C such that all parts are significant in producing the failure—nothing can be removed without making the failure disappear.

3. MINIMALITY OF TEST CASES

A simplified test case means the simplest possible web page that still reproduces the bug. If you remove any more characters from the file of the simplified test case, you no longer see the bug.
— *Mozilla BugAThon call*

How can one actually determine a minimal test case? Here comes bad news. Let there be some test case c consisting of $|c|$ changes (characters, lines, functions inserted) to the minimal input. Relying on *test* alone to determine minimality requires testing all $2^{|c|} - 1$ true subsets of c, which obviously has exponential complexity.[3]

What we can determine, however, is an *approximation*—for instance, a test case where every part on its own is still significant in producing the failure, but we do not check whether removing several parts at once might make the test case even smaller. Formally, we define this property as *1-minimality*, where n-minimality is defined as:

[2]POSIX 1003.3 also lists UNTESTED and UNSUPPORTED outcomes, which are of no relevance here.
[3]To be precise, Axiom 4 tells us the result of $test(\emptyset)$, such that only $2^{|c|} - 2$ subsets need to be tested, but this does not help much.

Minimizing Delta Debugging Algorithm

The *minimizing delta debugging algorithm* $ddmin(c)$ is

$$ddmin(c) = ddmin_2(c, 2) \quad \text{where}$$

$$ddmin_2(c, n) = \begin{cases} ddmin_2(c_i, 2) & \text{if } test(c_i) = ✘ \text{ for some } i \text{ ("reduce to subset")} \\ ddmin_2\big(\bar{c_i}, \max(n-1, 2)\big) & \text{else if } test(\bar{c_i}) = ✘ \text{ for some } i \text{ ("reduce to complement")} \\ ddmin_2\big(c, \min(|c|, 2n)\big) & \text{else if } n < |c| \text{ ("increase granularity")} \\ c & \text{otherwise ("done").} \end{cases}$$

where $c_1, \ldots, c_n \subseteq c$ such that $\bigcup c_i = c$, all c_i are pairwise disjoint, $\forall c_i \; (|c_i| \approx |c|/n)$, as well as $\bar{c_i} = c - c_i$.

The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c) = ✘ \wedge n \leq |c|$.

Figure 1: Minimizing delta debugging algorithm

Definition 6 (n-minimal test case) *A test case $c \subseteq C$ is n-minimal if*

$$\forall c' \subset c \left(|c| - |c'| \leq n \Rightarrow \big(test(c') \neq ✘\big)\right)$$

holds.

A failing test case c composed of $|c|$ lines would thus be *1-minimal* if removing any single line would cause the failure to disappear; likewise, it would be *3-minimal* if removing any combination of three or less lines would make it work again. If c is $|c|$-minimal, then c is minimal in the sense of Definition 5.

Definition 6 gives a first idea of what we should be aiming at. However, given, say, a 100,000 line test case, we cannot simply remove each individual line in order to minimize it. Thus, we need an effective algorithm to reduce our test case efficiently.

4. A MINIMIZING ALGORITHM

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.

— Brian Kernighan and Rob Pike, *The Practice of Programming*

What do humans do in order to minimize test cases? They use *binary search.* If c contains only one change, then c is minimal by definition. Otherwise, we *partition* c into two subsets c_1 and c_2 with similar size and test each of them. This gives us three possible outcomes:

Reduce to c_1. The test of c_1 fails—c_1 is a smaller test case.

Reduce to c_2. The test of c_2 fails—c_2 is a smaller test case.

Ignorance. Both tests pass, or are unresolved—neither c_1 nor c_2 qualify as possible simplifications.

In the first two cases, we can simply continue the search in the failing subset, as illustrated in Table 1. Each line of the diagram shows a configuration. A number i stands for an included change Δ_i; a dot stands for an excluded change. Change 7 is the minimal failing test case—and it is isolated in just a few steps.

Given sufficient knowledge about the nature of our input, we can certainly partition any test case into *two* subsets such that at least one of them fails the test. But what if this knowledge is insufficient, or not present at all?

Let us begin with the worst case: after splitting up c into subsets, all tests pass or are unresolved—ignorance is complete. All we know is that c as a whole is failing. How do we increase our chances of getting a failing subset?

- By testing *larger* subsets of C, we increase the chances that the test fails—the difference from C is smaller. On the other hand, a smaller difference means a slower progression—the test case is not halved, but reduced by a smaller amount.

- By testing *smaller* subsets of C, we get a faster progression in case the test fails. On the other hand, the chances that the test fails are smaller.

These specific methods can be combined by partitioning c into a *larger number of subsets* and testing each (small) c_i as well as its (large) complement $\bar{c_i}$—until each subset contains only one change, which gives us the best chance to get a failing test case. The disadvantage, of course, is that more subsets means more testing.

This is what can happen. Let n be the number of subsets $c_1, \ldots, c_n$. Testing each c_i and its complement $\bar{c_i} = c - c_i$, we have three possible outcomes (Figure 1):

Reduce to subset. If testing any c_i fails, then c_i is a smaller test case. Continue reducing c_i with $n = 2$ subsets.

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	?	
2	c_2	.	.	.	.	5	6	7	8	✘	
3	c_1	.	.	.	.	5	6	.	.	✔	
4	c_2	.	.	.	.	.	.	7	8	✘	
5	c_1	.	.	.	.	.	.	7	.	✘	Done
Result		.	.	.	.	.	.	7	.		

Table 1: Quick minimization of test cases

Step	c_i	Configuration								*test*	
1	$c_1 = \bar{c}_2$	1	2	3	4	.	.	.	.	**?**	Testing c_1, c_2
2	$c_2 = \bar{c}_1$	.	.	.	.	5	6	7	8	**?**	$\Rightarrow$ Increase granularity
3	c_1	1	2	.	.	.	.	.	.	**?**	Testing $c_1, \ldots, c_4$
4	c_2	.	.	3	4	.	.	.	.	✔	
5	c_3	.	.	.	.	5	6	.	.	✔	
6	c_4	.	.	.	.	.	.	7	8	**?**	
7	$\bar{c}_1$	.	.	3	4	5	6	7	8	**?**	Testing complements
8	$\bar{c}_2$	1	2	.	.	5	6	7	8	✘	$\Rightarrow$ Reduce to $c = \bar{c}_2$; continue with $n = 3$
9	c_1	1	2	.	.	.	.	.	.	**?***	Testing c_1, c_2, c_3
10	c_2	.	.	.	.	5	6	.	.	✔*	* same *test* carried out in an earlier step
11	c_3	.	.	.	.	.	.	7	8	**?***	
12	$\bar{c}_1$	.	.	.	.	5	6	7	8	**?**	Testing complements
13	$\bar{c}_2$	1	2	.	.	.	.	7	8	✘	$\Rightarrow$ Reduce to $c = \bar{c}_2$; continue with $n = 2$
14	$c_1 = \bar{c}_2$	1	2	.	.	.	.	.	.	**?***	Testing c_1, c_2
15	$c_2 = \bar{c}_1$	.	.	.	.	.	.	7	8	**?***	$\Rightarrow$ Increase granularity
16	c_1	1	.	.	.	.	.	.	.	**?**	Testing $c_1, \ldots, c_4$
17	c_2	.	2	.	.	.	.	.	.	✔	
18	c_3	.	.	.	.	.	.	7	.	**?**	
19	c_4	.	.	.	.	.	.	.	8	**?**	
20	$\bar{c}_1$	.	2	.	.	.	.	7	8	**?**	Testing complements
21	$\bar{c}_2$	1	.	.	.	.	.	7	8	✘	$\Rightarrow$ Reduce to $c = \bar{c}_2$; continue with $n = 3$
22	c_1	1	.	.	.	.	.	.	.	**?***	Testing $c_1, \ldots, c_3$
23	c_2	.	.	.	.	.	.	7	.	**?***	
24	c_3	.	.	.	.	.	.	.	8	**?***	
25	$\bar{c}_1$	.	.	.	.	.	.	7	8	**?**	Testing complements
26	$\bar{c}_2$	1	.	.	.	.	.	.	8	**?**	
27	$\bar{c}_3$	1	.	.	.	.	.	7	.	**?**	Done
Result		1	.	.	.	.	.	7	8		

Table 2: Minimizing a test case with increasing granularity

This reduction rule results in a classical "divide and conquer" approach. If one can identify a smaller part of the test case that is failure-inducing on its own, then this rule helps in narrowing down the test case efficiently.

Reduce to complement. If testing any $\bar{c}_i$ fails, then $\bar{c}_i$ is a smaller test case. Continue reducing $\bar{c}_i$ with $n - 1$ subsets.

Why do we continue with $n - 1$ and not two subsets here? Because splitting $\bar{c}_i$ into $n-1$ subsets means that the subsets of $\bar{c}_i$ are identical to the subsets c_i of c—in other words, every subset of c eventually gets tested. If we continued with two subsets from, say, $n = 32$, we would have to work our way down with $n = 2, 4, 8, \ldots$, but only with $n = 32$ would the next subset of c be tested.

Increase granularity. Otherwise (that is, no test failed), try again with $2n$ subsets. (Should $2n > |c|$ hold, try again with $|c|$ subsets instead, each containing one change.) This results in at most twice as many tests, but increases chances for failure.

The process is repeated until granularity can no longer be increased (that is, the next n would be larger than $|c|$). In this case, we have already tried removing every single change individually without further failures: the resulting change set is minimal.

As an example, consider Table 2, where the minimal test case consists of the changes 1, 7, and 8. Any test case that includes only a subset of these changes results in an unresolved test outcome; a test case that includes none of these changes passes the test.

We begin with partitioning the total set of changes in two halves—but none of them passes the test. We continue with granularity increased to 4 subsets (Step 3–6). When testing the complements, the set $\bar{c}_2$ fails, thus removing changes 3 and 4. We continue with splitting $\bar{c}_2$ in three subsets. The next three tests (Steps 9–11) have already been carried out and need not be repeated (marked with *). When testing $\bar{c}_2$ (Step 13), changes 5 and 6 can be eliminated. We increase granularity to 4 subsets and test each (Steps 16–19), before the last complement $\bar{c}_2$ (Step 21) eliminates change 2. Only changes 1, 7, and 8 remain; Steps 25–27 show that none of these changes can be eliminated. To minimize this test case, a total of 19 different tests was required.

We close with some formal properties of *ddmin*. First, *ddmin* eventually returns a 1-minimal test case:

Proposition 7 (*ddmin* minimizes) *For any $c \subseteq C$, ddmin(c) is 1-minimal in the sense of definition 6.*

PROOF. *According to the ddmin definition (Figure 1), ddmin(c) returns c only if $n \geq |c|$ and test$(\bar{c}_i) \neq$ ✘ for all $c_1, \ldots, c_n$. If $n \geq |c|$, then $|c_i| = 1$ and $|\bar{c}_i| = |c| - 1$. Since all subsets of $c' \subset c$ with $|c| - |c'| = 1$ are in $\{\bar{c}_1, \ldots, \bar{c}_n\}$ and test$(\bar{c}_i) \neq$ ✘ for all $\bar{c}_i$, the condition of definition 6 applies and c is 1-minimal.* $\square$

In the worst case, *ddmin* takes $3|c| + |c|^2$ tests:

Proposition 8 (*ddmin* complexity, worst case) *The number of tests carried out by ddmin(c) is $3|c| + |c|^2$ in the worst case.*

PROOF. *The worst case can be divided in two phases: First, every test is inconsistent until $n = |c|$ holds; then, testing only the last complement results in a failure until $n = 2$ holds.*

- *In the first phase, every test is inconsistent. This results in a re-invocation of $ddmin_2$ with a doubled number of subsets, until $|c_i| = 1$. The number of tests to be carried out is $2 + 4 + 8 + \cdots + 2|c| = 2|c| + |c| + \frac{|c|}{2} + \frac{|c|}{4} + \cdots = 4|c|$.*
- *In the second phase, the worst case is testing the* last *complement $\bar{c}_n$ fails, and $ddmin_2$ is re-invoked with $ddmin_2(\bar{c}_n, |c|-1)$. This results in $|c| - 1$ calls of ddmin, with two tests per call, or $2(|c|-1) + 2(|c|-2) + \cdots + 2 = 2 + 4 + 6 + \cdots + 2(|c|-1) = |c|(|c|-1) = |c|^2 - |c|$ tests.*

The overall number of tests is thus $4|c| + |c|^2 - |c| = 3|c| + |c|^2$. □

In practice, however, it is unlikely that an n-character input requires $3n + n^2$ tests. The "divide and conquer" rule of *ddmin* takes care of quickly narrowing down failure-inducing parts of the input:

Proposition 9 (*ddmin* complexity, best case) *If there is only one failure-inducing change $\Delta_i \in c$, and all test cases that include Δ_i cause a failure as well, then the number of tests t is limited by $t \leq 2 \log_2(|c|)$.*

PROOF. *Under the given conditions, Δ_i must always be in either c_1 or c_2, whose test will fail. Thus, the overall complexity is that of a binary search.* □

Whether this "best case" efficiency applies depends on our ability to break down the input into smaller chunks that result in determined (or better: failing) test outcomes. Consequently, the more knowledge about the structure of the input we have, the better we can identify possibly failure-inducing subsets, and the better is the overall performance of *ddmin*.

The surprising thing, however, is that even with *no knowledge about the input structure at all,* the *ddmin* algorithm has sufficient performance—at least in the case studies we have examined. This is illustrated in the following three sections.

5. CASE STUDY: GCC GETS A FATAL SIGNAL

None of us has time to study a large program to figure out how it would work if compiled correctly, much less which line of it was compiled wrong.
— Richard Stallman, *Using and Porting GNU CC*

Let us now turn to some real-life input. The C program in Figure 2 not only demonstrates some particular nasty aspects of the language, it also causes the GNU C compiler (GCC) to crash—at least, when using version 2.95.2 on Intel-Linux with optimization enabled. Before crashing, GCC grabs all available memory for its stack, such that other processes may run out of resources and die.[4] The latter can be prevented by limiting the stack memory available to GCC, but the effect remains:

```
#define SIZE 20

double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }

  return z[n];
}

void copy(double to[], double from[], int count)
{
   int n = (count + 7) / 8;
   switch (count % 8) do {
      case 0: *to++ = *from++;
      case 7: *to++ = *from++;
      case 6: *to++ = *from++;
      case 5: *to++ = *from++;
      case 4: *to++ = *from++;
      case 3: *to++ = *from++;
      case 2: *to++ = *from++;
      case 1: *to++ = *from++;
   } while (---n > 0);

   return mult(to, 2);
}

int main(int argc, char *argv[])
{
   double x[SIZE], y[SIZE];
   double *px = x;

   while (px < x + SIZE)
      *px++ = (px - x) * (SIZE + 1.0);

   return copy(y, x, SIZE);
}
```

Figure 2: The `bug.c` program that crashes GNU CC

```
$ (ulimit -H -s 256; gcc -O bug.c)
gcc: Internal compiler error:
     program cc1 got fatal signal 11
$ _
```

The GCC error message (and the resulting core dump) help GCC maintainers only; as ordinary users, we must now narrow down the failure-inducing input in `bug.c`—and *minimize* `bug.c` in order to file in a bug report.

In the case of GCC, the minimal test input is the empty input. For the sake of simplicity, we modeled a *change* as the *insertion of a single character.* This means that

- each change c_i becomes the i-th character of `bug.c`
- C becomes the entire failure-inducing input `bug.c`
- partitioning C means partitioning the input into parts.

No special effort was made to exploit syntactic or semantic knowledge about C programs; consequently, we expected a large number

[4]The authors deny any liability for damage caused by repeating this experiment.

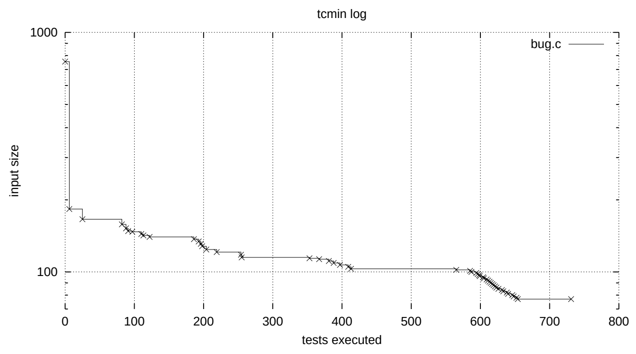

Figure 3: Minimizing GCC input `bug.c`

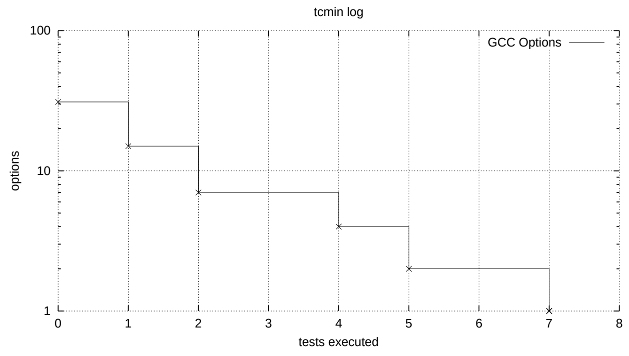

Figure 5: Minimizing GCC options

of test cases to be invalid C programs.

To minimize `bug.c`, we implemented the *ddmin* algorithm of Figure 1 into our WYNOT prototype[5]. The *test* procedure would create the appropriate subset of `bug.c`, feed it to GCC, return ✘ iff GCC had crashed, and ✔ otherwise. The results of this WYNOT run are shown in Figure 3.

After the first two tests, WYNOT has already reduced the input size from 755 characters to 377 and 188 characters, respectively—the test case now only contains the *mult* function. Reducing *mult*, however, takes time: only after 731 more tests (and 34 seconds)[6] do we get a test case that can not be minimized any further. Only 77 characters are left:

t(double z[],int n){int i, j;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0] + 0)$;}return $z[n]$;}

This test case is 1-minimal—no single character can be removed without removing the failure. Even every single superfluous whitespace has been removed, and the function name has shrunk from *mult* to a single t. (At least, we now know that neither whitespace nor function name were failure-inducing!)

Figure 4 shows an excerpt from the `bug.c` test log. (The character □ indicates an omitted character with regard to the minimized input.) We see how the *ddmin* algorithm tries to remove every single change (= character) in order to minimize the input even further—but every test results in a syntactically invalid program.

t(double z[],int n){int i, j;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[\square] + 0)$;}return $z[n]$;}
t(double z[],int n){int i, j;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0\square + 0)$;}return $z[n]$;}
t(double z[],int n){int i, j;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0]\ \square\ 0)$;}return $z[n]$;}
t(double z[],int n){int i, j;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0] + \square)$;}return $z[n]$;}
t(double z[],int n){int i, j;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0] + 0\square$;}return $z[n]$;}
t(double z[],int n){int i, j;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0] + 0)\square$}return $z[n]$;}

t(double z[],int n){int i, j;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0] + 0)$;}return $z[n]$;}

Figure 4: Excerpt from the `bug.c` test log

As GCC users, we can now file this in as a minimal bug report. But where in GCC does the failure actually occur? We already know that the failure is associated with optimization. Could it be possible to influence optimization in a way that the failure disappears?

The GCC documentation lists 31 options that can be used to influence optimization on Linux, shown in Table 3. It turns out that applying *all of these options* causes the failure to disappear:

```
$ gcc -O -ffloat-store -fno-default-inline \
   -fno-defer-pop ...-fstrict-aliasing bug.c
$ _
```

This means that some option(s) in the list *prevent* the failure. We can use test case minimization in order to find the preventing option(s). This time, each c_i stands for a GCC option from Table 3. Since we want to find an option that *prevents* the failure, the *test* outcome is inverted: *test* returns ✔ if GCC crashes and ✘ if GCC works fine.

This WYNOT run is a straight-forward "divide and conquer" search, shown in Figure 5. After 7 tests (and less than a second), the single option *–ffast-math* is found which prevents the failure:

```
$ gcc -O -ffast-math bug.c
$ _
```

Unfortunately, the *–ffast-math* option is a bad candidate for working around the failure, because it may alter the semantics of the program. We remove *–ffast-math* from the list of options and make another WYNOT run. Again after 7 tests, it turns out the option *–fforce-addr* also prevents the failure:

–ffloat-store	*–fno-default-inline*	*–fno-defer-pop*
–fforce-mem	*–fforce-addr*	*–fomit-frame-pointer*
–fno-inline	*–finline-functions*	*–fkeep-inline-functions*
–fkeep-static-consts	*–fno-function-cse*	*–ffast-math*
–fstrength-reduce	*–fthread-jumps*	*–fcse-follow-jumps*
–fcse-skip-blocks	*–frerun-cse-after-loop*	*–frerun-loop-opt*
–fgcse	*–fexpensive-optimizations*	*–fschedule-insns*
–fschedule-insns2	*–ffunction-sections*	*–fdata-sections*
–fcaller-saves	*–funroll-loops*	*–funroll-all-loops*
–fmove-all-movables	*–freduce-all-givs*	*–fno-peephole*
–fstrict-aliasing		

Table 3: GCC optimization options

[5] WYNOT = "Worked Yesterday, NOt Today"

[6] All times were measured on a Linux PC with a 500 MHz Pentium III processor. The time given is the CPU user time of our WYNOT prototype as measured by the UNIX kernel; it includes all spawned child processes (such as the GCC run in this example).

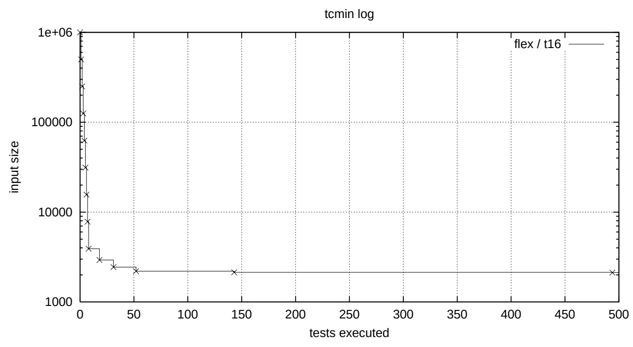

Figure 6: Minimizing FLEX fuzz input

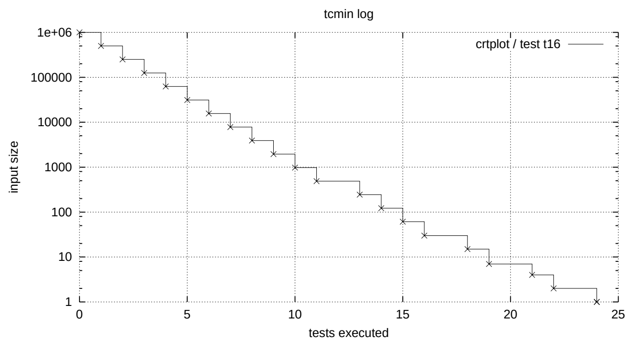

Figure 7: Minimizing CRTPLOT fuzz input

```
$ gcc -O -fforce-addr bug.c
$ _
```

Are there any other options that prevent the failure? Running GCC with the remaining 29 options shows that the failure is still there; so it seems we have identified all failure-preventing options. And this is what we can send to the GCC maintainers:

1. The minimal test case
2. "The failure occurs only with optimization."
3. "*–ffast-math* and *–fforce-addr* prevent the failure."

Still, we cannot identify a place in the GCC code that causes the problem. On the other hand, we have identified as many *failure circumstances* as we can. In practice, program maintainers can easily enhance their automated regression test suites such that the failure circumstances are automatically simplified for any failing test case.

6. CASE STUDY: MINIMIZING FUZZ

If you understand the context in which a problem occurs, you're more likely to solve the problem completely rather than only one aspect of it.

— Steve McConnell, *Code Complete*

In a classical experiment [6, 7], Bart Miller and his team 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% of the basic programs crashed or went into infinite loops when being fed with fuzz input.

We wanted to know how well the *ddmin* algorithm performs in minimizing the fuzz input sequences. We examined a subset of the UNIX utilities listed in Miller's paper: NROFF (format documents for display), TROFF (format documents for typesetter), FLEX (fast lexical analyzer generator), CRTPLOT (graphics filter for various plotters), UL (underlining filter), and UNITS (convert quantities).

We set up 16 different fuzz inputs, differing in size (10^3 to 10^6 characters) and content (whether all characters or only printable characters were included, and whether NUL characters were included or not). As shown in Table 4, Miller's results still apply—at least on Sun's Solaris 2.6 operating system: out of $6 \times 16 = 96$ test runs, the utilities crashed 42 times (43%).

We applied our WYNOT tool in all 42 cases to minimize the failure-inducing fuzz input. Table 5 shows the resulting input sizes; Table 6 lists the number of tests required.[7] Depending on the crash cause, the programs could be partitioned into two groups:

- The first group of programs shows obvious *buffer overrun* problems.
  - FLEX, the most robust utility, crashes on sequences of 2,121 or more non-newline and non-NUL characters (t_{14}–t_{15}).
  - UL crashes on sequences of 516 or more printable non-newline characters (t_5–t_8, t_{13}–t_{16}).
  - UNITS crashes on sequences of 77 or more 8-bit characters (t_2–t_4 and t_{11}–t_{12}).

  Figure 6 shows the first 500 tests of the WYNOT run for FLEX and t_{16}. After 494 tests, the remaining size of 2,122 characters is already close to the final size; however, it takes more than 10,000 further tests to eliminate one more character.

- The second group of programs appears vulnerable to *random commands*.
  - NROFF and TROFF crash
    - on *malformed commands* like `"\\D^J%0F"`[8] (NROFF, t_6), and
    - on *8-bit input* such as `"\302\n"` (TROFF, t_1)
  - CRTPLOT crashes on the one-letter inputs `"t"` (t_1) and `"f"` (t_5, t_9, t_{13}–t_{16}).

  The WYNOT run for CRTPLOT and t_{16} is shown in Figure 7. It takes 24 tests to minimize the fuzz input of 10^6 characters to the single failure-inducing character.

Again, all test runs can be (and have been) entirely automated. This allows for *massive automated stochastic testing*, where programs are fed with fuzz input in order to reveal defects. As soon as a failure is detected, input minimization can generalize the large fuzz input to a minimal bug report.

[7]Table 6 also includes *repeated tests* which have been carried out in earlier steps. On the average, the number of actual (non-repeated) tests is 30% smaller.

[8]All input is shown in C string notation.

Name	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}
Input size	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6
Character range	all				printable				all				printable			
NUL characters	yes				yes				no				no			
NROFF	✘^{S}	✘^{S}	✘^{S}	✘^{S}	–	✘^{A}	✘^{A}	✘^{A}	✘^{S}	✘^{S}	✘^{S}	✘^{S}	–	–	–	–
TROFF	–	✘^{S}	✘^{S}	✘^{S}	–	✘^{A}	✘^{A}	✘^{S}	–	–	✘^{S}	✘^{S}	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	✘^{S}	✘^{S}	✘^{S}
CRTPLOT	✘^{S}	–	–	–	✘^{S}	–	–	–	✘^{S}	–	–	–	✘^{S}	✘^{S}	✘^{S}	✘^{S}
UL	–	–	–	–	✘^{S}	✘^{S}	✘^{S}	✘^{S}	–	–	–	–	✘^{S}	✘^{S}	✘^{S}	✘^{S}
UNITS	–	✘^{S}	✘^{S}	✘^{S}	–	–	–	–	–	–	✘^{S}	✘^{S}	–	–	–	–

"–" = test passed (✔), ✘^{S}= Segmentation Fault, ✘^{A}= Arithmetic Exception

Table 4: Test outcomes of UNIX utilities subjected to fuzz input

Name	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}
Input size	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6
Character range	all				printable				all				printable			
NUL characters	yes				yes				no				no			
NROFF	2	2	2	2	–	7	7	7	2	2	2	2	–	–	–	–
TROFF	–	3	3	3	–	7	7	7	–	–	3	3	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	2121	2121	2121
CRTPLOT	1	–	–	–	1	–	–	–	1	–	–	–	1	1	1	1
UL	–	–	–	–	516	516	516	516	–	–	–	–	516	516	516	516
UNITS	–	77	77	77	–	–	–	–	–	–	77	77	–	–	–	–

Table 5: Size of minimized failure-inducing fuzz input

7. CASE STUDY: MOZILLA CANNOT PRINT

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.

— *Mozilla BugAThon call*

As a last case study, we wanted simplify a real-world Mozilla test case and thus contribute to the Mozilla BugAThon. A search in Bugzilla, the Mozilla bug database, shows us bug #24735, reported by *anantk@yahoo.com*:

> Ok the following operations cause mozilla to crash consistently on my machine
>
> → 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 this case, the Mozilla input consists of two items: The *sequence of input events*—that is, the succession of mouse motions, pressed keys, and clicked buttons—and the *HTML code* of the erroneous WWW page. We used the XLAB *capture/replay* tool [13] to run Mozilla while capturing all user actions and logging them to a file. We could easily reproduce the error, creating an XLAB log with 711 recorded X events. Our WYNOT tool would now use XLAB to *replay* the log and feed Mozilla with the recorded user actions, thus automating Mozilla execution.

In a first run, we wanted to know whether all actions in the bug report were actually necessary. We thus subjected the log to test case minimization, in order to find a *failure-inducing minimum of user actions.* Out of the 711 X events, only 95 were related to user actions—that is, moving the mouse pointer, pressing or releasing the mouse button, and pressing or releasing a key on the keyboard. These 95 user actions were subjected to minimization.

The results of this run are shown in Figure 9. After 82 test runs (or 21 minutes), only 3 out of 95 user actions are 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.)

User actions removed 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 this is irrelevant in producing the failure.[9]

Since the user actions can hardly be further generalized, we turn our attention to another input source–the failure-inducing HTML code. The original *Search for bug* page has a length of 39094 characters or 896 lines. In order to minimize the HTML code, we chose a *hierarchical* approach: In a first run, we wanted to minimize the *number of lines* (that is, each c_i was identified with a line); in a later run, we wanted to minimize the failure-inducing line(s) according to single characters.

[9]It is relevant, though, that the mouse button be pressed before it is released.

Name	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}
Input size	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6	10^3	10^4	10^5	10^6
Character range	all				printable				all				printable			
NUL characters	yes				yes				no				no			
NROFF	55	41	60	39	–	156	153	243	17	22	27	54	–	–	–	–
TROFF	–	84	73	100	–	156	153	22493	–	–	50	42	–	–	–	–
FLEX	–	–	–	–	–	–	–	–	–	–	–	–	–	11589	17960	10619
CRTPLOT	15	–	–	–	15	–	–	–	16	–	–	–	14	17	23	24
UL	–	–	–	–	7138	7012	6058	7090	–	–	–	–	2434	3455	3055	2307
UNITS	–	662	623	626	–	–	–	–	–	–	630	221095	–	–	–	–

Table 6: Number of required test runs

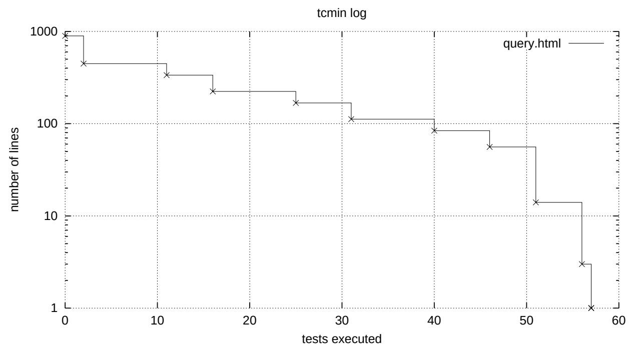

Figure 8: Minimizing Mozilla HTML input

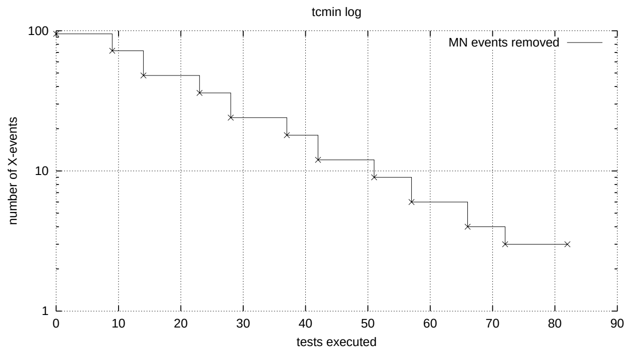

Figure 9: Minimizing Mozilla user actions

The results of the *lines* run are shown in Figure 8. After 57 test runs, the *ddmin* algorithm minimizes the original 896 lines to a 1-line input:

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

This is the HTML input which causes Mozilla to crash when being printed. As in the GCC example of Section 5, the actual failure-inducing input is very small. Further minimization[10] reveals that the attributes of the `SELECT` tag are not relevant for reproducing the failure, either, such that the single input

```
<SELECT>
```

already suffices for reproducing the failure. Overall, we obtain the following self-contained minimized bug report:

→ Create a HTML page containing "`<SELECT>`"

→ Load the page and print it using *Alt+P* and *Print*.

→ The browser crashes with a segmentation fault.

As long as the bug reports can be reproduced, this minimization procedure can easily be repeated automatically with the 5595 other bugs listed in the Bugzilla database[11]. 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.

[10]This minimization was done by hand. We apologize.

[11]as of 14 Feb 2000, 14:00 GMT

8. RELATED WORK

When you have two competing theories which make exactly the same predictions, the one that is simpler is the better.

— *Occam's Razor*

As stated in the introduction, we are unaware of any other technique that would automatically simplify test cases to determine failure-inducing input. One important exception is the simplification of test cases which have been *artificially produced.* In [11], Don Slutz describes how to stress-test databases with generated SQL statements. After a failure has been produced, the test cases had to be simplified—after all, a failing 1,000-line SQL statement would not be taken seriously by the database vendor, but a 3-line statement would. This simplification was realized simply by undoing the earlier production steps and testing whether the failure still occurred,

In general, delta debugging determines circumstances that are relevant for producing a failure (in our case, parts of the program input.) In the field of automated debugging, such failure-inducing circumstances have almost exclusively been understood as failure-inducing *statements* during a program execution. The most significant method to determine statements relevant for a failure is *program slicing*—either the static form obtained by program analysis [14, 12] or the dynamic form applied to a specific run of the program [1, 3].

The strength of analysis is that several potential failure causes can be eliminated due to lack of data or control dependency. This does not suffice, though, to check whether the remaining potential causes are relevant or not for producing a given failure. Only by experiment (that is, testing) can we prove that some circumstance is relevant—by showing that there is some alteration of the circumstance that makes the failure disappear. When it comes to concrete failures, program analysis and testing are complementary: analysis disproves causality, and testing proves it.

It would be nice to see how far systematic testing and program analysis could work together and whether delta debugging could be used to determine failure-inducing statements as well. Just as determining which parts of the input were relevant in producing the failure, delta debugging could determine the failure-relevant statements in the program. *Critical slicing* [2] is a related approach which is test-based like delta debugging; additional data flow analysis is used to eliminate circumstantial positives.

The *ddmin* algorithm presented in this paper is an alternative to the original delta debugging algorithm dd^+ presented in [15]. Like *ddmin*, dd^+ takes a set of changes and minimizes it according to a given test; in [15], these changes affected the program code and were obtained by comparing two program versions.

The main differences between *ddmin* and dd^+ are:

- dd^+ determines the minimal difference between a failing and a non-failing configuration, while *ddmin* minimizes the difference between a failing and an empty configuration.
- dd^+ is not well-suited for failures induced by a large combination of changes. In particular, dd^+ does not guarantee a 1-minimal subset, which is why it is not suited for minimizing test cases.
- dd^+ assumes *monotony:* that is, whenever $test(c) = ✔$ holds, then $test(c') = ✔$ holds for every subset of c as well. This assumption, which was found to be useful for changes to program code, gave dd^+ a better performance when most tests produced determinate results.

We recommend *ddmin* as a general replacement for dd^+. To exploit monotony in *ddmin*, one can make $test(c)$ return ✔ whenever a superset of c has already passed the test.

9. FUTURE WORK

If you get all the way up to the group-signed T-Shirt, you can *qualify for a stuffed animal as well by doing 12 more.*
— *Mozilla BugAThon call*

Our future work will concentrate on the following topics:

Domain-specific simplification methods. Knowledge about the input structure can very much enhance the performance of the *ddmin* algorithm. For instance, valid program inputs are frequently described by *grammars*; it would be nice to rely on such grammars in order to exclude syntactically invalid input right from the start. Also, with a formal input description, one could replace input by smaller *alternate input* rather than simply cutting it away. In the GCC example, one could try to replace arithmetic expressions by constants, or program blocks by no-ops; HTML input could be reduced according to HTML structure rules.

Optimization. In general, the abstract description of the *ddmin* algorithm leaves a lot of flexibility in the actual implementation and thus provides "hooks" for several domain-specific optimizations:

- The implementation can choose how to *partition* c into subsets c_i. This is the place where knowledge about the structure of the input comes in handy.
- The implementation can choose *which subset to test first.* Some subsets may be more likely to cause a failure than others.
- The implementation can choose whether and how to handle *multiple independent failure-inducing inputs*—that is, the case where there are several subsets c_i with $test(c_i) = ✘$. Options include
  - to continue with the first failing subset,
  - to continue with the smallest failing one, or
  - to simplify each individual failing subset.

  Our implementation currently goes for the first failing subset only and thus reports only one subset. The reason is economy: it is wiser to fix the first failure before checking for further similar failures.

Program analysis. So far, we have treated all tested programs as black boxes, not referring to source code at all. However, there are several *program analysis* methods available that can help in relating input to a specific failure, or that can simply tell us which parts of the input are related (and can thus be changed in one run) and which others not. A simple *dynamic slice* of the failing test case can tell us which input actually influenced the program and which input never did. The combination of input-centered and execution-centered debugging methods remains to be explored.

Maximizing passing test cases. Right now, *ddmin* makes no distinction between passing and unresolved tests. There are several settings, however, where such a distinction may be useful, and where we could minimize the *difference* between a passing and a failing test—not only by minimizing the failure-inducing input, but also by *maximizing the passing input.* We expect that such a two-folded approach pinpoints the failure faster and more precisely.

Other failure-inducing circumstances. Changing the input of the program is only one means to influence its execution. As stated in Section 2, a Δ_i can stand for any change in the circumstances that influences the execution of the program. We will thus research whether delta debugging is applicable to further failure-inducing circumstances such as executed statements, control predicates or thread schedules.

10. CONCLUSION

Debugging is still, as it was 30 years ago, a matter of trial and error.
— Henry Lieberman, *The Debugging Scandal*

We have shown how the *ddmin* algorithm simplifies failure-inducing input, based on an automated testing procedure. The method can be (and has been) applied in a number of settings, finding failure-inducing parts in the program invocation (GCC options), in the program input (GCC, Fuzz, and Mozilla input), or in the sequence of user interactions (Mozilla user actions).

We recommend that automated test case simplification be an integrated part of automated testing. Each time a test fails, delta debugging could be used to simplify the circumstances of the failure. Given sufficient testing resources and a reasonable choice of changes Δ_i that influence the program execution, the *ddmin* algorithm presented in this paper provides a simplification that is straight-forward and easy to implement.

In practice, testing and debugging typically come in pairs. However, in debugging research, testing has played a very minor role. This is surprising, because re-testing a program under changed circumstances is a common debugging approach. Delta debugging does nothing but to automate this process. Eventually, we expect that several debugging tasks can in fact be stated as search and minimization problems, based on automated testing—and thus be solved automatically.

More details on the case studies listed in this paper can be found in [4]. Further information on delta debugging, including the full WYNOT implementation, is available at

`http://www.fmi.uni-passau.de/st/dd/` .

Acknowledgements. Mirko Streckenbach provided helpful insights on UNIX internals. Tom Truscott pointed us to the GCC error. Holger Cleve, Jens Krinke and Gregor Snelting provided valuable comments on earlier revisions of this paper. Special thanks go to the anonymous reviewers for their constructive comments.

11. REFERENCES

[1] H. Agrawal and J. Horgan. Dynamic program slicing. *SIGPLAN Notices*, 6:246–256, 1990.

[2] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In S. J. Zeil, editor, *Proc. of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, volume 21(3) of *ACM Software Engineering Notes*, pages 121–134, San Diego, California, USA, Jan. 1996.

[3] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In Nierstrasz and Lemoine [10], pages 303–321.

[4] R. Hildebrandt. Minimierung fehlerverursachender Eingaben. Diploma thesis, Technical University of Braunschweig, Germany, Apr. 2000. In German.

[5] IEEE, New York. *Test Methods for Measuring Conformance to POSIX*, 1991. ANSI/IEEE Standard 1003.3-1991. ISO/IEC Standard 13210-1994.

[6] B. P. Miller, L. Fredrikson, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, Dec. 1990.

[7] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisted: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Science Department, Nov. 1995.

[8] Mozilla web site. http://www.mozilla.org/.

[9] Mozilla web site: The Gecko BugAThon. http://www.mozilla.org/newlayout/bugathon.html.

[10] O. Nierstrasz and M. 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*, Toulouse, France, Sept. 1999. Springer-Verlag.

[11] D. R. Slutz. Massive stochastic testing of SQL. In A. Gupta, O. Shmueli, and J. Widom, editors, *Proc. of 24rd International Conference on Very Large Data Bases (VLDB'98), New York City, New York, USA*, pages 618–622. Morgan Kaufmann, Aug. 1998.

[12] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[13] M. Vertes. Xlab—a tool to automate graphical user interfaces. *Linux Weekly News*, May 1998. Archived as http://lwn.net/980528/a/xlab.html.

[14] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[15] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In Nierstrasz and Lemoine [10], pages 253–267.