

# *Simplifying Failure-Inducing Input*

*Ralf Hildebrandt and Andreas Zeller*

*ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*
*Portland, Oregon, August 23, 2000*

# *The Mozilla BugAThon*

*Mozilla*—Netscape's open source web browser project

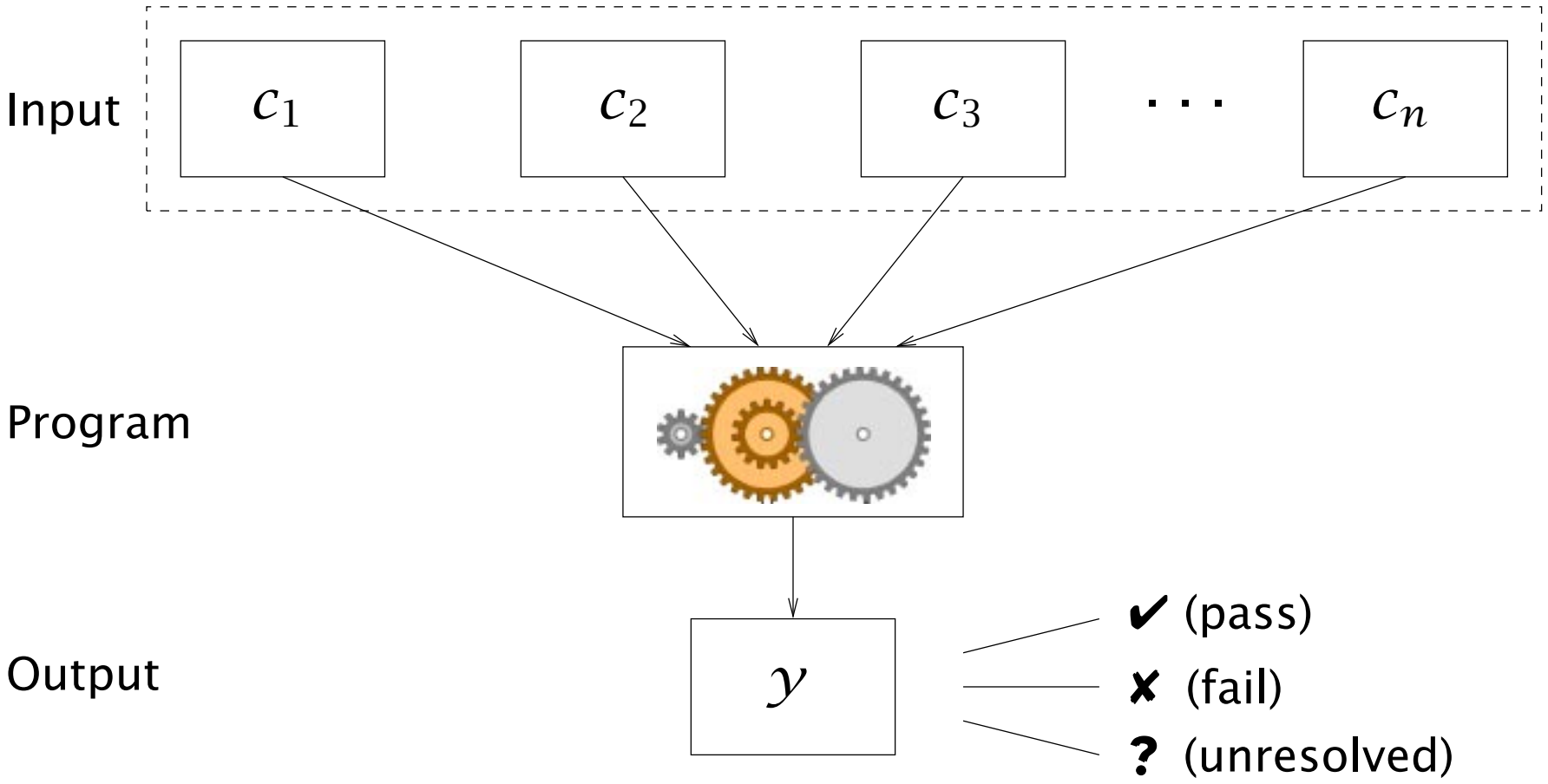
Maintained by dozens of Netscape engineers and 100s of volunteers

In February 2000: ~5,500 open bugs in the Bugzilla database

*Mozilla BugAThon*—call for volunteers who would *simplify test cases:*

| Pledge Level | Reward |
|---|---|
| 5 bugs | invitation to the Gecko launch party |
| 10 bugs | the invitation, plus an attractive Gecko stuffed animal |
| 12 bugs | same, but animal autographed by the Father of Gecko |
| 15 bugs | the invitation, plus a Gecko T-shirt |
| 17 bugs | same, but T-shirt signed by the grateful engineer |
| 20 bugs | same, but T-shirt signed by the whole raptor team |

# *Failure-Inducing Circumstances*




Which of the circumstances $c_i$ are the causes for $y$?

# *Differing Circumstances*

Assumption—two program runs under differing circumstances:

- one passes the test (✔)—e.g. on some trivial or empty input
- the other one fails (✘)—the one we're interested in

Assumption: a gradual *transition* between these two runs.

We denote the *differences* between these program runs by a *set of changes* $C = \{\Delta_1, \ldots, \Delta_n\}$—i.e. changes applied to the circumstances.

A $\Delta_i$ can stand for:

- the insertion of a single character
- the deletion of a line
- the insertion of a substructure...

# Tests

A *test case* is a subset of changes $c \subseteq C$.

Let $test : 2^C \rightarrow \{✔, ✘, ?\}$ be a function which checks a test case.

Three possible outcomes:

- The test passes (✔)
- The test fails (✘)
- The test outcome is unresolved (?)

Axioms:

$$test(\emptyset) = ✔ \quad \text{(“cause absent, effect absent”)}$$
$$test(C) = ✘ \quad \text{(“cause present, effect present”)}$$

# *Minimal Test Cases*

Our goal: a *minimal test case* $c \subseteq C$

If $c$ is *minimal*, the failure does not occur in any subset:

$$\forall c' \subset c \; (test(c') \neq \text{✘})$$

Problem: One must test all $2^{|c|} - 1$ subsets of $c$.

Pragmatic approach: a *1-minimal* test case

No single $\Delta_i$ can be omitted without causing the failure to disappear:

$$\forall c' \subset c \left( |c| - |c'| \leq 1 \Rightarrow (test(c') \neq \text{✘}) \right)$$

"If you remove any more characters from the file of the simplified test case, you no longer see the bug." (Mozilla BugAThon)

# *A Minimizing Algorithm*

Basic pattern: Start by removing large chunks, try smaller ones later. . .

. . . until the automated test fails—and then repeat with smaller subset.

- Guarantees 1-minimality (every subset will eventually be tested)
- Best efficiency for small failure-inducing input

# *A Minimizing Algorithm (2)*

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

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

$$ddmin_2(c, n) = \begin{cases} ddmin_2(c_i, 2) & \text{if } \exists i \cdot test(c_i) = \text{✘} \\ ddmin_2(c - c_i, \max(n-1, 2)) & \text{if } \exists i \cdot test(c - c_i) = \text{✘} \\ ddmin_2(c, \min(|c|, 2n)) & \text{if } n < |c| \\ c & \text{otherwise} \end{cases}$$

where $c = \bigcup c_i$ with $c_i$ pairwise disjoint and $\forall c_i \cdot (|c_i| \approx |c|/n)$.

Number of tests: $|c|^2 + 3|c|$ in worst case, $\log_2 |c|$ in best case.

# Example: GCC Dumps Core

```
#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);
}
```

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

# Example: GCC Dumps Core (2)

| Step | Test case | *test* |
|---|---|---|
| 1 | #define SIZE 20\n double **mult**(double $z$[],int $n$) { ... } □ | ✘ |
| 2 | #define SIZE 20\n □ | ✔ |
| 3 | □ double **mult**(double $z$[],int $n$) { ... } | ✘ |
| 4 | double **mult**(double $z$[],int $n$) { int i, j; i = 0; □ } | ✔ |
| 5 | double **mult**(double $z$[],int $n$) { □ for($j$ = 0; $j$ < $n$; $j$++) { ... } ... } | ? |
| ⋮ | ⋮ | ⋮ |
| 839 | **t**(double $z$[],int $n$){int $i$,$j$;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[$□] + 0);}return $z[n]$;} | ? |
| 840 | **t**(double $z$[],int $n$){int $i$,$j$;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0$□ + 0);}return $z[n]$;} | ? |
| 841 | **t**(double $z$[],int $n$){int $i$,$j$;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0] $□ 0);}return $z[n]$;} | ? |
| 842 | **t**(double $z$[],int $n$){int $i$,$j$;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0] + □);}return $z[n]$;} | ? |
| 843 | **t**(double $z$[],int $n$){int $i$,$j$;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0] + 0□;}return $z[n]$;} | ? |
| 844 | **t**(double $z$[],int $n$){int $i$,$j$;for(;;){$i = i + j + 1$;$z[i] = z[i] * (z[0] + 0)□}return $z[n]$;} | ? |
| ⋮ | ⋮ | ⋮ |

Minimal input found after 857 tests:

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

# *Example: Minimizing Fuzz*

Classical experiment: UNIX tools fed with *fuzz* input (10,000 random characters). Most crash.

Minimizing input reveals causes:

| Program | Minimized Input | *test* runs |
|---|---|---|
| `flex` – lexical analyzer | ⟨2121 characters⟩ | 11589 |
| `ul` – do underlining | ⟨516 characters⟩ | 3055 |
| `nroff` – format documents | "\302\n" | 60 |
| `plot` – graphics filter | "f" | 17 |

(Tests carried out on a Sun Solaris 2.6 machine)

# *Example: Mozilla Cannot Print*

Mozilla 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

# *Mozilla Cannot Print—Minimizing User Actions*

X11 Capture/Replay tool recorded 95 user actions (mouse motions, key presses, etc.)

Delta Debugging simplified these user actions to 3 relevant ones (82 test runs / 21 minutes):

1. Press the *P* key while the *Alt* key is held. (Invoke the *Print* dialog.)
2. Press *mouse button 1* on the *Print* button (Arm the *Print* button.)
3. Release *mouse button 1*. (Start printing.)

Everything else is irrelevant—including releasing the *P* key.

# *Mozilla Cannot Print—Minimizing HTML*

The original *Search for bug* page has a length of 896 lines.

Delta Debugging simplified this page to a single line (57 test runs):

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

Minimization by characters minimized the line even further.

New, simplified 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.

# *Future Work: Integrating Analysis*

Basic idea: reduce large number of tests by additional knowledge

**Structure knowledge** can be a big help in decomposing input:

- Decompose GCC input according to C syntax
- Decompose TROFF input by lines
- Decompose HTML input according to HTML syntax...

**Relating the input to the output** (by means of program analysis) helps in finding good candidates for causality.

## *Future Work: Alternate Circumstances*

Besides program input, one may consider alternate circumstances that affect program execution:

- Changes to the program code (Zeller 1999)
- Executed functions
- Performed schedules
- Taken branches. . .

Delta debugging can separate all these into relevant and irrelevant circumstances—hopefully with the help of program analysis.

# *Future Work: Open Issues*

When is a run considered a failure?

- Too few details ⇒ more false positives
- Too many details ⇒ larger number of circumstances

# *Future Work: Open Issues (2)*

Delta debugging minimizes problems of the kind

$$\Delta_1 \wedge \Delta_2 \wedge \cdots \wedge \Delta_n \Leftrightarrow y$$

But problems may also look like

$$\Delta_i \vee \Delta_j \Leftrightarrow y \quad \text{or} \quad \neg\Delta_i \Leftrightarrow y \quad \text{or} \ldots$$

The “simplest” causality is in fact the shortest algorithm $f$ that computes $y = f(\Delta_1, \ldots, \Delta_n)$

# *Causes and Events*

The *cause* of any event is a preceding event without which the event in question would not have occurred.

How to demonstrate causality? John Stuart Mill (1806–1873):

- **Method of agreement**—Effect present when cause present
- **Method of difference**—Effect absent when cause absent
- **Method of concomitant variation**—Both agreement and difference (stronger)

Causality cannot be demonstrated without experimentation!

# *Causality is the Key*

Example program:

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

Output:

```
a = 0
```

What does this say about b?



```
float a;
```

# *Conclusion*

Delta debugging. . .

- automatically simplifies failure-inducing circumstances
- proves causality by experimentation
- requires large number of tests (but analysis can help!)

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