Isolating Failure Causes

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken
Causes and Alternate Worlds

Failing World

Causes for Failure

Common Context

Causes for Non-Failure

Passing World
The Narrowing Process

1. Set up hypothesis

2a. Hypothesis confirmed
2b. Hypothesis rejected

3. Repeat as needed

Alternate World

Reduced difference = more specific cause

Passing World

Initial Difference = Initial Cause

Failing World

or

Reduced difference = more specific cause

Hypothesis confirmed

Hypothesis rejected

Repeat as needed
Simplifying HTML Input

Idea: Apply *Divide and Conquer* to simplify HTML pages

Simplified bug report: **Printing** `<SELECT>` **crashes.**
Simplifying vs. Isolating

Problem: To simplify the entire input can be expensive

Alternative approach: We do not simplify the entire input, but the *difference* with respect to a *working input*.

Simplifying

Isolating

Larger context – but fewer tests and smaller causes
Isolating a HTML difference

<table>
<thead>
<tr>
<th>#</th>
<th>Mozilla input</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>✗</td>
</tr>
<tr>
<td>4</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>✗</td>
</tr>
<tr>
<td>7</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>✓</td>
</tr>
</tbody>
</table>

Isolated difference: the “<” in “<SELECT>”.

Isolating requires 7 tests, simplifying 26.
Simplification vs. Isolation

**Simplification**

- make *each part* of the simplified test case relevant
- removing *any part* makes the failure go away

**Isolation**

- find *one* relevant part of the test case
- removing *this particular part* makes the failure go away

Both simplification and isolation can be handled by delta debugging.
Recalling \textit{ddmin}

\[ \text{ddmin}(c_x) = \text{ddmin}'(c_x, 2) \quad \text{where} \]

\[
\text{ddmin}'(c'_x, n) = \begin{cases}
\text{ddmin}'(\nabla_i, \max(n - 1, 2)) & \text{if } \exists i \in \{1, \ldots, n\} \\
\text{ddmin}'(c'_x, \min(2n, |c_x|)) & \text{if } 2n < |c_x| \\
c'_x & \text{otherwise}
\end{cases}
\]

with \( c'_x = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n, \nabla_i = c'_x \setminus \Delta_i, \) and

\[ \forall \Delta_i, \Delta_j : \Delta_i \cap \Delta_j = \emptyset \land |\Delta_i| \approx |\Delta_j|. \]

The \textit{ddmin} algorithm must be \textit{extended} to compute differences.
A new Algorithm

Let us try to formalize our issues.

Again, we have $c^\checkmark$, $c^\times$, $C$, etc. as defined for $ddmin$.

Our goal is to find two sets $c^\prime\checkmark$ and $c^\prime\times$ such that

- $\emptyset = c^\checkmark \subseteq c^\prime\checkmark \subset c^\prime\times \subseteq c^\times$ holds and
- the difference $\Delta = c^\prime\times - c^\prime\checkmark$ is 1-minimal.

$\Delta$ is 1-minimal if

$$\forall \delta_i \in \Delta \cdot \text{test}(c^\prime\checkmark \cup \{\delta_i\}) \neq \checkmark \land \text{test}(c^\prime\times - \{\delta_i\}) \neq \times$$

holds.
Extending *ddmin*

We must extend *ddmin* such that it works on *two sets at a time*:

- The failing test case $c'_\times$ which is to be *minimized* (initially, $c'_\times = c_\times$ holds), and
- The passing test case $c'_\checkmark$ which is to be *maximized* (initially, $c'_\checkmark = c_\checkmark = \emptyset$ holds).
A Binary Search Approach

Basic idea:

- We split the difference $\Delta = c'_x - c'_\checkmark$ into two subsets $\Delta_1$ and $\Delta_2$.
  
  $\Delta = \Delta_1 \cup \Delta_2$, $\Delta_1 \cap \Delta_2 = \emptyset$, and $|\Delta_1| \approx |\Delta_2|$ holds.

- We test two configurations:
  
  - $c'_x \setminus \Delta_1 = c'_\checkmark \cup \Delta_2$ and
  - $c'_x \setminus \Delta_2 = c'_\checkmark \cup \Delta_1$
**Possible Outcomes**

Starting with $c'_\checkmark = c_\checkmark$, $c'_\times = c_\times$; $\Delta = c'_\times - c'_\checkmark = \Delta_1 \cup \Delta_2$.

<table>
<thead>
<tr>
<th>Test</th>
<th>Outcome</th>
<th>New $c'_\checkmark$</th>
<th>New $c'_\times$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c'<em>\times \setminus \Delta_1 = c'</em>\checkmark \cup \Delta_2$</td>
<td>✔</td>
<td>$c'_\checkmark$</td>
<td>$c'_\times \setminus \Delta_1$</td>
</tr>
<tr>
<td>$c'<em>\times \setminus \Delta_1 = c'</em>\checkmark \cup \Delta_2$</td>
<td>✗</td>
<td>$c'_\checkmark \cup \Delta_2$</td>
<td>$c'_\times$</td>
</tr>
<tr>
<td>$c'<em>\times \setminus \Delta_2 = c'</em>\checkmark \cup \Delta_1$</td>
<td>✔</td>
<td>$c'_\checkmark$</td>
<td>$c'_\times \setminus \Delta_2$</td>
</tr>
<tr>
<td>$c'<em>\times \setminus \Delta_2 = c'</em>\checkmark \cup \Delta_1$</td>
<td>✗</td>
<td>$c'_\checkmark \cup \Delta_1$</td>
<td>$c'_\times$</td>
</tr>
</tbody>
</table>

Classical binary search with $O(\log_2 |\Delta|)$ tests.
The *ddbin* Algorithm

Given: \( \text{test}, c_\vee, c_\times \cdot c_\vee \subseteq c_\times \land \text{test}(c_\vee) = \checkmark \land \text{test}(c_\times) = \times \).

Goal: \( c'_\vee, c'_\times = \text{ddbin}(c_\vee, c_\times) \) such that \( c_\vee \subseteq c'_\vee \subseteq c'_\times \subseteq c_\times \), \( \text{test}(c'_\vee) = \checkmark \), \( \text{test}(c'_\times) = \times \)
and each element of \( \Delta = c'_\times \setminus c'_\vee \) is relevant for the failure.

Let \( \Delta = c'_\times \setminus c'_\vee = \Delta_1 \cup \Delta_2 \) in

\[
\text{ddbin}(c_\vee, c_\times) = \text{ddbin}'(c_\vee, c_\times) \text{ where }
\]

\[
\text{ddbin}'(c'_\times, c'_\vee) = \begin{cases} 
(c'_\vee, c'_\times) & \text{if } |\Delta| = 1 \\
\text{ddbin}'(c'_\vee, c'_\times \cup \Delta_2) & \text{if } \text{test}(c'_\vee \cup \Delta_2) = \times \\
\text{ddbin}'(c'_\times \setminus \Delta_2, c'_\vee) & \text{if } \text{test}(c'_\times \setminus \Delta_2) = \checkmark \\
\text{ddbin}'(c'_\vee, c'_\times \cup \Delta_1) & \text{if } \text{test}(c'_\vee \cup \Delta_1) = \times \\
\text{ddbin}'(c'_\times \setminus \Delta_1, c'_\vee) & \text{if } \text{test}(c'_\times \setminus \Delta_1) = \checkmark 
\end{cases}
\]

(Note that \( c'_\times \setminus \Delta_1 = c'_\vee \cup \Delta_2 \) and \( c'_\times \setminus \Delta_2 = c'_\vee \cup \Delta_1 \) hold.)

Classical binary search!
## ddbin on Mozilla input

<table>
<thead>
<tr>
<th>#</th>
<th>Mozilla Input</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code> ✔</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code> ✗</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code> ✔</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code> ✗</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code> ✔</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code> ✔</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code> ✔</td>
<td></td>
</tr>
</tbody>
</table>
# Unresolved Test Outcomes

Problem: *ddbin* does not handle *unresolved* test outcomes!

<table>
<thead>
<tr>
<th>Step</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#define SIZE 20 ... double <strong>mult</strong>(...) { ... }</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>#define SIZE 20</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>double <strong>mult</strong>(...) { ... }</td>
<td>✗</td>
</tr>
<tr>
<td>4</td>
<td>double <strong>mult</strong>(...) { int i, j; i = 0; }</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>double <strong>mult</strong>(...) { for(...) { ...} ... }</td>
<td>?</td>
</tr>
</tbody>
</table>
Unresolved Test Outcomes (2)

The *more we change* some input which has a *resolved* test outcome (✔ or ✗),

- the *faster* the progress in narrowing the difference, but
- the *higher* are the chances of unresolved outcomes (?)..

If we apply *smaller changes* to the input,

- the chance to get an unresolved outcome is *smaller*, but
- the *progress* is smaller, too!

We need a *compromise* between these two approaches!
Unresolved Test Outcomes (3)

Basic idea:

1. Start with *few & large changes* first
2. If all alternatives are unresolved, apply *more & smaller changes.*

This is achieved by splitting the initial $\Delta$ not into *two* subsets, but into an *increasing* number of subsets—as in *ddmin*!

Thus, we have to *merge* the binary search of the *ddbin* algorithm with the arbitrary number of subsets as in *ddmin.*
General Delta Debugging

Given: test, c✔, c✘ • c✔ ⊆ c✘ ∧ test(c✔) = ✔ ∧ test(c✘) = ✘.

Goal: c’✔, c’✘ = dd(c✔, c✘) such that c✔ ⊆ c’✔ ⊆ c’✘ ⊆ c✘, test(c’✔) = ✔, test(c’✘) = ✘
and each element of ∆ = c’✘ \ c’✔ is relevant for the failure.

Let ∆ = c’✘ \ c’✔ = Δ₁ ∪ · · · ∪ Δₙ in

dd(c✔, c✘) = dd’(c✔, c✘, 2) where

\[
\begin{align*}
dd’(c’✔, c’✔ \cup Δ_i, 2) & \quad \text{if } \exists i \cdot \text{test}(c’✔ \cup Δ_i) = ✘, \\
& \text{if } \exists i \cdot \text{test}(c’✘ \setminus Δ_i) = ✔, \\
& \text{if } \exists i \cdot \text{test}(c’✔ \cup Δ_i) = ✔, \\
& \text{if } 2n < |Δ|, \\
& \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
& dd’(c’✘ \setminus Δ_i, c’✘, \max(n – 1, 2)) & \quad \text{if } \exists i \cdot \text{test}(c’✔ \cup Δ_i) = ✘, \\
& dd’(c’✔ \cup Δ_i, c’✘, \max(n – 1, 2)) & \quad \text{if } \exists i \cdot \text{test}(c’✘ \setminus Δ_i) = ✘, \\
& dd’(c’✔, \min(2n, |Δ|)) & \quad \text{if } 2n < |Δ|, \\
& (c’✔, c’✘) & \quad \text{otherwise}
\end{align*}
\]
**dd vs. ddbin vs. ddmin**

*dd* is the most general of all Delta Debugging algorithms:

- If *test* returns ✔ for c✔ only, and ? in all other cases, then *dd* is equivalent to *ddmin*.
- If *test* never returns ?, then *dd* is equivalent to *ddbin* (= binary search)

Consequence 1: You only need to know about *dd*. Period.

Consequence 2: *We must avoid unresolved test outcomes as good as we can* (e.g. by adding syntactic or semantic knowledge, as in simplification)
Application: Code Changes

Date: Fri, 31 Jul 1998 15:11:05 -0500
From: Brian Kahne <bkahne@ibmoto.com>
To: DDD Bug Reports <bug-ddd@gnu.org>
Subject: Problem with DDD and GDB 4.17

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

Yesterday, my Program Worked

Assumption: The failure is caused by one of the changes between “yesterday” and “today”.

Goal: Finding and examining this failure-inducing change.

Procedure: Delta Debugging
Trouble Ahead

In case of GDB, we have an enormous change:

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

and so on for a total of 178,200 lines.
Trouble Ahead (2)

Large changes are not the only source of trouble:

✘ Granularity. A single logical change can affect thousands of lines of code—but only a few lines may be responsible for the failure.
Example: integration of large third-party changes

✘ Interference. There can be multiple failure-inducing changes that cause the failure only when applied together.
Example: integration of parallel development lines

✘ Inconsistency. The generated configuration may be inconsistent—we do not know whether the failure occurs.
Example: change conflict, construction failure, crash

All these are handled by delta debugging.
Isolating the GDB Change

DIFF split into 8721 changes; 370s/test on 400 MHz PC

![Delta Debugging Log](image)

The failure-inducing code change is:

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."
---
> "Set argument list to give program being debugged when it is started."
**Application: Thread Schedules**

The behavior of a multi-threaded program can depend on the *thread schedule*:

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>open(&quot;.htpasswd&quot;)</td>
<td>open(&quot;.htpasswd&quot;)</td>
</tr>
<tr>
<td></td>
<td>read(...)</td>
<td>read(...)</td>
</tr>
<tr>
<td></td>
<td>modify(...)</td>
<td>modify(...)</td>
</tr>
<tr>
<td></td>
<td>write(...)</td>
<td>write(...)</td>
</tr>
<tr>
<td></td>
<td>close(...)</td>
<td>close(...)</td>
</tr>
</tbody>
</table>

Thread Switch
The behavior of a multi-threaded program can depend on the thread schedule:

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td><code>open(&quot;.htpasswd&quot;)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>read(...)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>modify(...)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>write(...)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>close(...)</code></td>
<td></td>
</tr>
<tr>
<td>✔️</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌️</td>
<td><code>open(&quot;.htpasswd&quot;)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>read(...)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>modify(...)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>write(...)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>close(...)</code></td>
<td></td>
</tr>
<tr>
<td>❌️</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Application: Thread Schedules
Application: Thread Schedules

The behavior of a multi-threaded program can depend on the thread schedule:

Thread switches and schedules are *nondeterministic*: Bugs are *hard to reproduce* and *hard to isolate*!
Recording and Replaying Runs

DEJAVU captures and replays program runs deterministically:

![Diagram showing recorded schedule and replay]

Allows simple reproduction of schedules and induced failures
**Differences between Schedules**

Using DEJAVU, we can consider the schedule as an *input* which determines whether the program passes or fails.

![Diagram showing replay with ✔ and ✘ symbols]
**Differences between Schedules**

Using DEJAVU, we can consider the schedule as an *input* which determines whether the program passes or fails.

The *difference* between schedules is relevant for the failure:

A *small* difference can pinpoint the failure cause.
Finding Differences

- We start with runs ✔ and ✗.
- We determine the differences $\Delta_i$ between thread switches $t_i$:
  - $t_1$ occurs in ✔ at “time” 254
  - $t_1$ occurs in ✗ at “time” 278
  - The difference $\Delta_1 = |278 - 254|$ induces a statement interval: the code executed between “time” 254 and 278
  - Same applies to $t_2$, $t_3$, etc.

Our goal: Narrow down the difference such that only a small relevant difference remains, pinpointing the root cause.
Isolating Relevant Differences

We use Delta Debugging to isolate the relevant differences.

Delta Debugging applies subsets of differences to:

- The entire difference $\Delta_1$ is applied
- Half of the difference $\Delta_2$ is applied
- $\Delta_3$ is not applied at all

DEJAVU executes the debuggee under this generated schedule; an automated test checks if the failure occurs
**The Isolation Process**

Delta Debugging systematically narrows down the difference.
A Real Program

We examine Test #205 of the SPEC JVM98 Java test suite: a raytracer program depicting a dinosaur.

Program is single-threaded—the multi-threaded code is commented out.

To test our approach,

- we make the raytracer program multi-threaded again
- we introduce a simple race condition
- we implement an automated test that would check whether the failure occurs or not
- we generate random schedules until we obtain both a passing schedule (✔) and a failing schedule (✘)
Passing and Failing Schedule

We obtain two schedules with 3,842,577,240 differences, each moving a thread switch by ±1 “time” unit.
Narrowing Down the Failure Cause

Delta Debugging isolates one single difference after 50 tests:
The Root Cause of the Failure

```java
public class Scene {
    ...

    private static int ScenesLoaded = 0;
    (more methods...)

private int LoadScene(String filename) {
    int OldScenesLoaded = ScenesLoaded;
    (more initializations...)
    infile = new DataInputStream(...);
    (more code...)
    ScenesLoaded = OldScenesLoaded + 1;
    System.out.println("" +
                     ScenesLoaded + " scenes loaded.");
    ...
```

```
Consequence

Still, processor speed doubles almost every 18 months (Moore’s Law).

Consequence: We can now afford approaches that were way too expensive only a few years ago.

Currently, the computer spends 99.9% of its time just waiting for the programmer to move the mouse.

We can exploit this to have

- Expensive program analysis
- Automated testing
- Automated debugging
In contrast to simplification, *isolation* finds only one relevant part of the test case; removing *this particular part* makes the failure go away.

Isolation is *much more efficient* than simplification.
Concepts (2)

- The general Delta Debugging algorithm \textit{dd} extends \textit{ddmin} to isolate failure-inducing differences.

- \textit{dd} becomes an efficient binary search as soon as there are no unresolved test outcomes

- Delta Debugging can be applied to \textit{arbitrary circumstances} of the program run:
  - Program input
  - Program code
  - Program environment (i.e., the thread schedule)

as long as there is an automated test and a passing configuration to compare with.