Isolating Failure Causes

Andreas Zeller
Isolating Causes

Actual world

Alternate world

Test

Mixed world
Isolating Causes

Actual world

Alternate world

Test

Mixed world

“How can we automate this?”

+ 1.0
Simplifying Input

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

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

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

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

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

Input

Failure Cause

…”
Isolating Input

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

Difference narrowed down

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

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

<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
Isolating Input

<select name="priority" multiple size=7>
  <option value="1">Failure Cause</option>
</select>
Isolating

Input

Failure Cause
Finding Causes

- minimal input
- minimal context

- minimal difference
- common context
Configuration

Circumstance \( \delta \)

All circumstances

\[ C = \{ \delta_1, \delta_2, \ldots \} \]

Configuration \( c \subseteq C \)

\[ c = \{ \delta_1, \delta_2, \ldots, \delta_n \} \]
Tests

Testing function

\[ test(c) \in \{\checkmark, \times, ?\} \]

Initial configurations

\[ test(c\checkmark) = \checkmark \]
\[ test(c\times) = \times \]
Minimal Difference

Goal: Subsets $c^\prime_x$ and $c^\prime_\checkmark$

$\emptyset = c_\checkmark \subseteq c^\prime_\checkmark \subset c^\prime_x \subseteq c_x$

Difference

$\Delta = c^\prime_x \setminus c^\prime_\checkmark$

Difference is 1-minimal

$\forall \delta_i \in \Delta \cdot test(c^\prime_\checkmark \cup \{\delta_i\}) \neq \checkmark \land test(c^\prime_x \setminus \{\delta_i\}) \neq \times$
Algorithm Sketch

• Extend $ddmin$ such that it works on two sets at a time – $c'_x$ and $c'_✓$

• Compute subsets

$$\Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n = \Delta = c'_x \setminus c'_✓$$

• For each subset, test

  • the addition $c'_✓ \cup \Delta_i$

  • the removal $c'_x \setminus \Delta_i$
### Test Outcomes

<table>
<thead>
<tr>
<th>Condition</th>
<th>Positive Outcome</th>
<th>Negative Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{test}(c'_x \setminus \Delta_i)$</td>
<td>$c'_x := c'_x \setminus \Delta_i$</td>
<td>$c' := c'_x \setminus \Delta_i$</td>
</tr>
<tr>
<td>$\text{test}(c'_x \cup \Delta_i)$</td>
<td>$c'_x := c'_x \cup \Delta_i$</td>
<td>$c' := c'_x \cup \Delta_i$</td>
</tr>
<tr>
<td>otherwise</td>
<td>increase granularity</td>
<td></td>
</tr>
</tbody>
</table>

*most valuable outcomes*
**dd in a Nutshell**

\[ dd(c_\checkmark, c_\times) = (c'_\checkmark, c'_\checkmark) \quad \Delta = c'_\times \setminus c'_\checkmark \quad \text{is 1-minimal} \]

\[ dd(c_\checkmark, c_\times) = dd'(c_\checkmark, c_\times, 2) \]

\[ dd'(c'_\checkmark, c'_\times, n) = \]

\[
\begin{cases}
(c'_\checkmark, c'_\times) \\
dd'(c'_\times \setminus \Delta_i, c'_\times, 2) \\
dd'(c'_\times, c'_\checkmark \cup \Delta_i, 2) \\
\min(2n, |\Delta|)) \\
\min(2n, |\Delta|)) \\
\end{cases}
\]

if \(|\Delta| = 1\)

if \(\exists i \in \{1..n\} \cdot \text{test}(c'_\times \setminus \Delta_i) = 1\)

if \(\exists i \in \{1..n\} \cdot \text{test}(c'_\times \cup \Delta_i) = \checkmark\)

else if \(\exists i \in \{1..n\} \cdot \text{test}(c'_\times \cup \Delta_i) = \times\)

else if \(\exists i \in \{1..n\} \cdot \text{test}(c'_\times \setminus \Delta_i) = \checkmark\)

else if \(\exists i \in \{1..n\} \cdot \text{test}(c'_\times \setminus \Delta_i) = \times\)

else if \(n < |\Delta|\) ("increase granularity")

otherwise
def dd(c_pass, c_fail):
    n = 2
    while 1:
        delta = listminus(c_fail, c_pass)
        deltas = split(delta, n); offset = 0; j = 0
        while j < n:
            i = (j + offset) % n
            next_c_pass = listunion(c_pass, deltas[i])
            next_c_fail = listminus(c_fail, deltas[i])
            if test(next_c_fail) == FAIL and n == 2:
                c_fail = next_c_fail; n = 2; offset = 0; break
            elif test(next_c_fail) == PASS:
                c_pass = next_c_fail; n = 2; offset = 0; break
            elif test(next_c_pass) == FAIL:
                c_fail = next_c_pass; n = 2; offset = 0; break
            elif test(next_c_fail) == FAIL:
                c_fail = next_c_fail; n = max(n - 1, 2); offset = i; break
            elif test(next_c_pass) == PASS:
                c_pass = next_c_pass; n = max(n - 1, 2); offset = i; break
            else:
                j = j + 1
        if j >= n:
            if n >= len(delta):
                return (delta, c_pass, c_fail)
            else:
                n = min(len(delta), n * 2)
Properties

number of tests $t$ — worst case:

$$t = |\Delta|^2 + 7|\Delta| \quad \text{where} \quad \Delta = c_x \setminus c_\check{x}$$

number of tests $t$ — best case
(no unresolved outcomes):

$$t \leq \log_2(\Delta)$$

size of difference — no unresolved outcomes

$$|c'_x \setminus c'_\check{x}| = 1$$
### Applications

<table>
<thead>
<tr>
<th>Input</th>
<th>Code Changes</th>
<th>Schedules</th>
</tr>
</thead>
</table>


Isolating Input

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

Failure
Simplification: 48 tests

Isolation: 5 tests

19
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. [...]
Version Differences

New version

Program fails

Program works

Causes

Old version
What was Changed

$ 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 178,200 lines (8,721 locations)
Challenges

- **Granularity** – within some large change, only a few lines may be relevant
- **Interference** – some (later) changes rely on other (earlier) changes
- **Inconsistency** – some changes may have to be combined to produce testable code

Delta debugging handles all this
General Plan

• Decompose diff into changes per location (= 8,721 individual changes)

• Apply subset of changes, using PATCH

• Reconstruct GDB; build errors mean unresolved test outcome

• Test GDB and return outcome
Isolating Changes

- Result after 98 tests (≈ 1 hour)
The Failure Cause

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."

- Documentation becomes GDB output
- DDD expects Arguments, but GDB outputs Argument list
The change on the following file is failure-inducing: `src/org/apache/commons/lang/StringUtils.java`

The line was added: `one line was added.`

The file `Constants.java` was added to project `commons-lang-1_copy_524290`

```java
public class Constants {

    public static final int ZERO = 1;

    // More code...
}
```
Optimizations

• History – group changes by creation time
• Reconstruction – cache several builds
• Grouping – according to scope
• Failure Resolution – scan error messages for possibly missing changes
**Thread Schedules**

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<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>

**A’s updates get lost!**
Record + Replay

DEJAVU captures and replays program runs deterministically:

- DEJAVU recorded schedule
- DEJAVU allows simple reproduction of schedules and induced failures

```plaintext
x = 45
y = 39
z = 67

x = 45
y = 39
z = 67

x = 45
y = 39
z = 67

x = 45
y = 39
z = 67
```
Schedules as Input

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

The schedule difference causes the failure!
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.
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.

Diagram:
- Initial state: 100
- Applied $\Delta_1$: 110
- Half of $\Delta_2$: 112
- $\Delta_3$: No change
- Final state: 110
Isolating Differences

Delta Debugging systematically narrows down the difference.

Dejavu replays the generated schedule.

Test outcome
Example: Raytracer

- Raytracer program from Spec JVM98 suite
- Injected a simple \textit{race condition}
- Set up \textit{automated test} + random schedules
- Obtained \textit{passing} and \textit{failing} schedule
- 3,842,577,240 differences, each moving a thread switch by $\pm 1$ \textit{yield point} (time unit)
Isolating Schedules

no unresolved outcomes: complexity is $O(\log_2 n)$
The Failure Cause

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.");
...
}
General Issues

• How do we choose the alternate world?
• How do we decompose the configuration?
• How do we know a failure is the failure?
• How do we disambiguate multiple causes?
• How do I get to the defect?
 Concepts

★ To isolate failure causes automatically, use
  • an *automated test case*
  • a means to *narrow down the difference*
  • a *strategy* for proceeding.

★ One possible strategy is Delta Debugging.
Concepts (2)

★ Delta Debugging can isolate failure causes

• in the (general) input

• in the version history

• in thread schedules

★ Every such cause implies a fix – but not necessarily a correction.