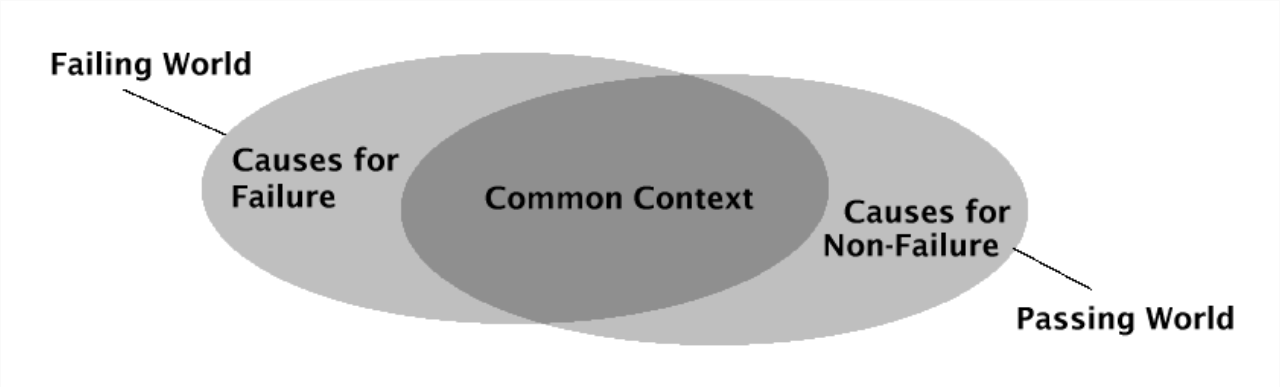
Simplifying Test Cases

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Causes and Alternate Worlds



Divide and Conquer

Divide and conquer (from latin “divide et impera”):
The basic principle of reducing a *large* problem into *smaller* subproblems.

General pattern:

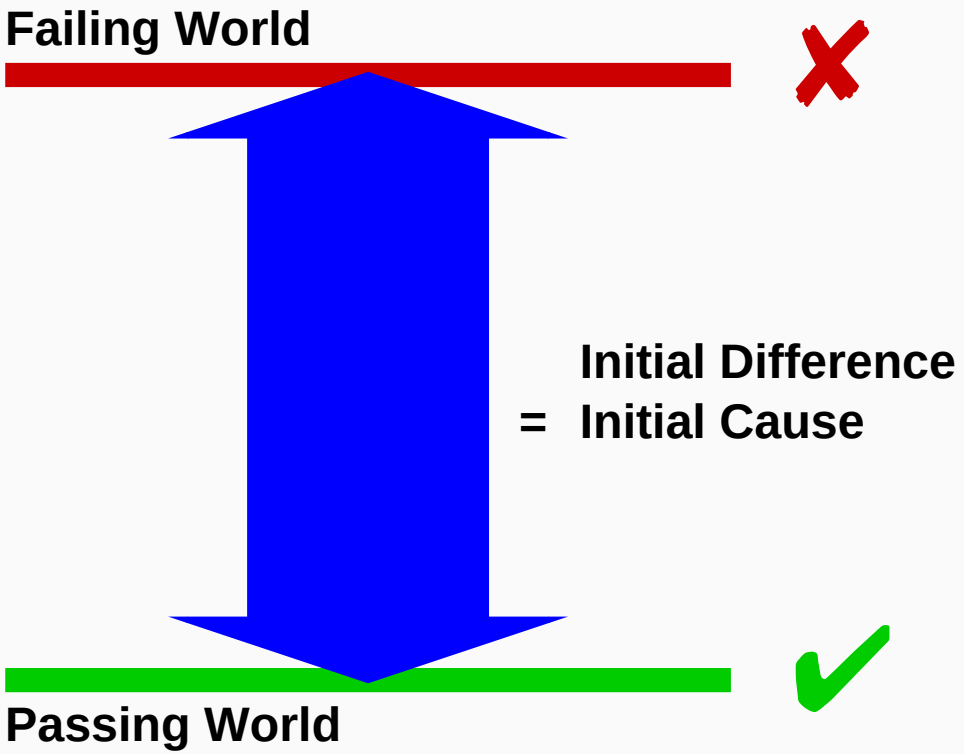
- Start with some *initial* difference (= initial cause)
- *Decompose* this difference into smaller differences
- *Test* the resulting hypotheses to see whether the smaller differences are more specific failure causes.



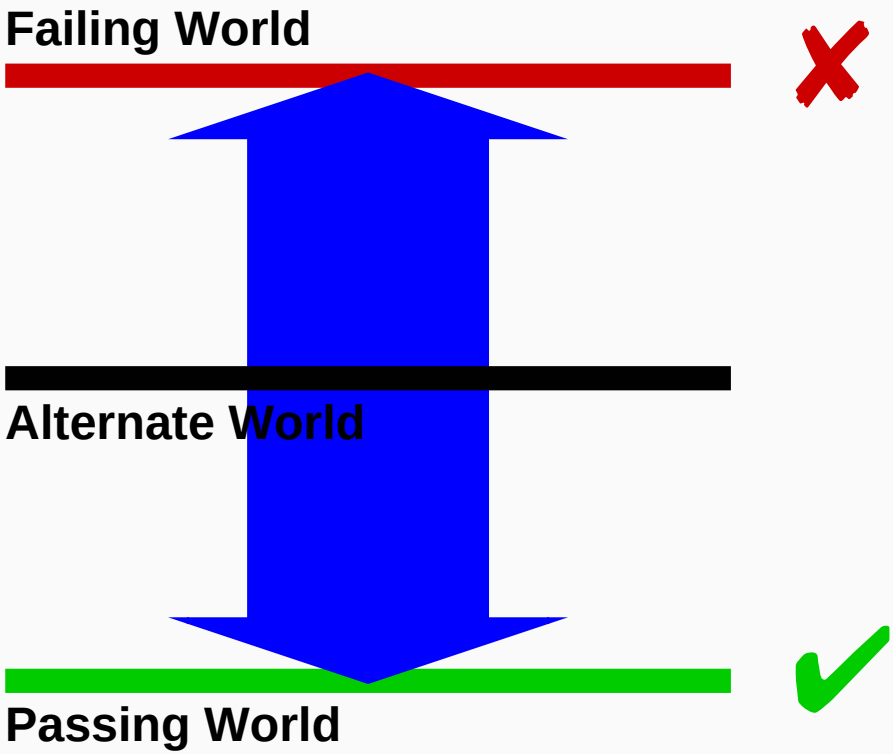




The Initial Difference

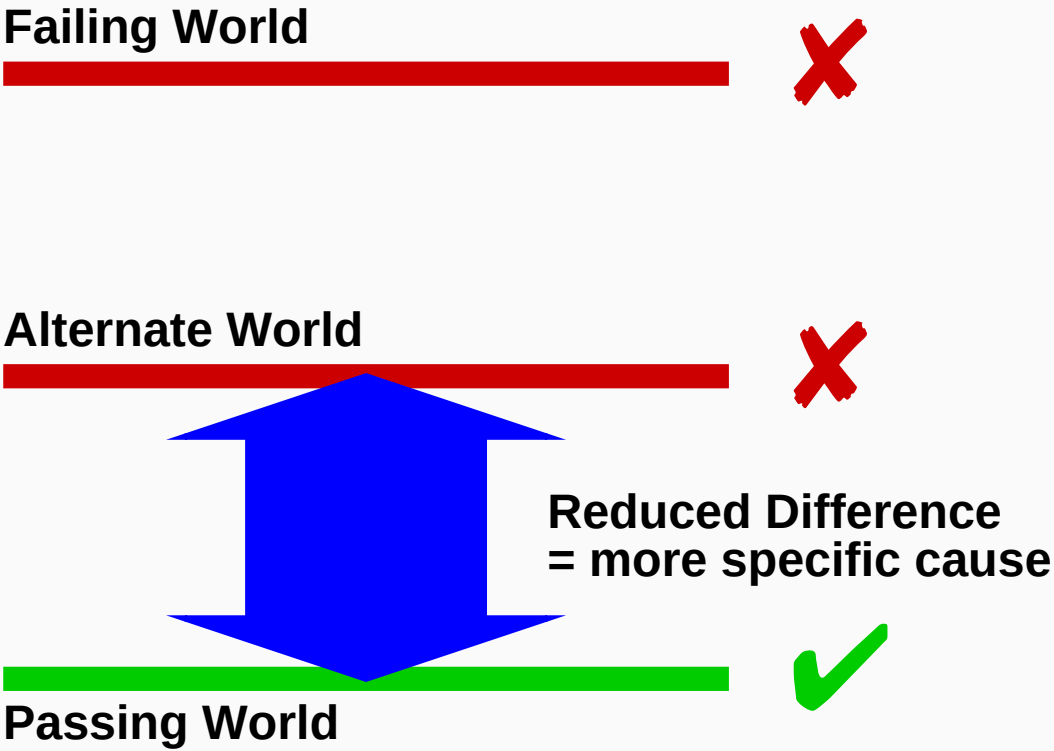


Setting up an Alternate World

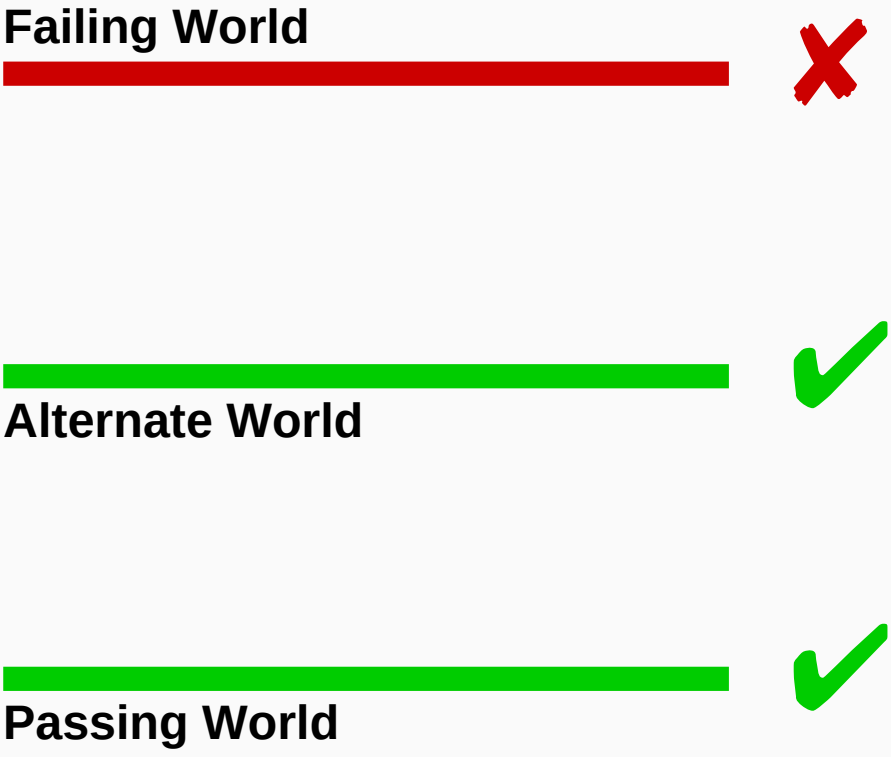


Testing the Alternate World

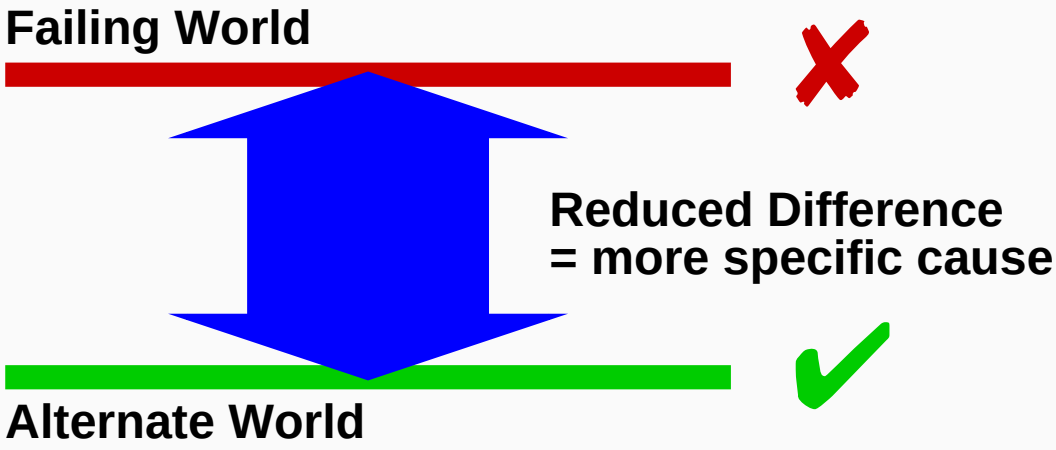
Narrowing the Difference



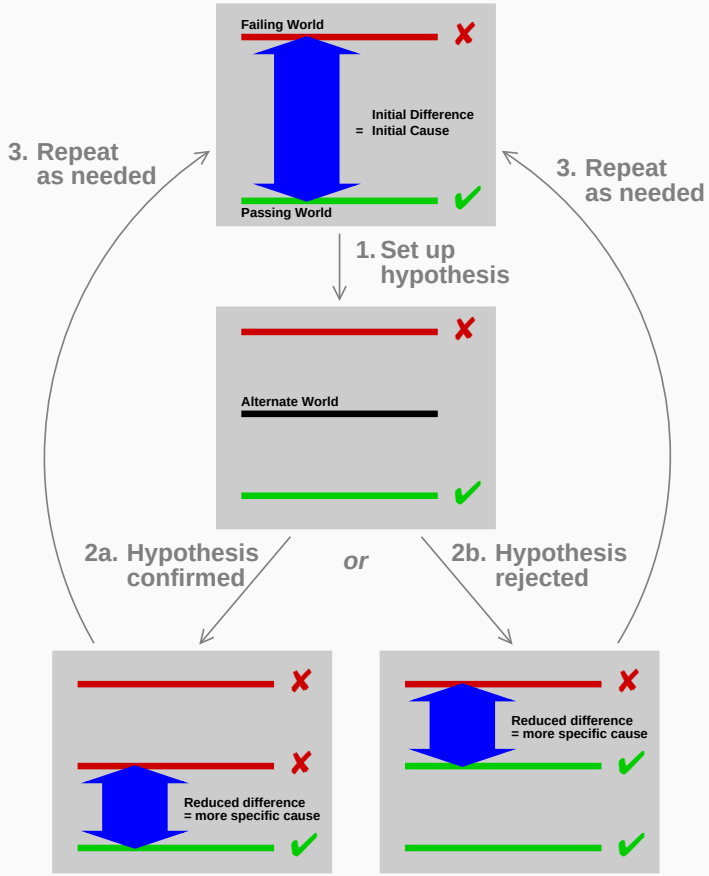
Alternate Test Outcome



Narrowing the Difference



The Narrowing Process



Mozilla Crashes when Printing

After reproducing a failure comes *simplifying* the test case
⇒ reduce a test case to *relevant* details only.

Example —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
```

What is *relevant* in this problem report?







The Mozilla Input


```
<td align=left valign=top>
<SELECT NAME="op_sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows
95">Windows 95<OPTION VALUE="Windows 98">Windows 98<OPTION VALUE="Windows
ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac System
7.5">Mac System 7.5<OPTION VALUE="Mac System 7.6.1">Mac System 7.6.1<OPTION
VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac
System 9.x<OPTION VALUE="MacOS X">MacOS X<OPTION VALUE="Linux">Linux<OPTION
VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION
VALUE="HP-UX">HP-UX<OPTION VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION
VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION VALUE="OSF/1">OSF/1<OPTION
VALUE="Solaris">Solaris<OPTION VALUE="SunOS">SunOS<OPTION
VALUE="other">other</SELECT>

</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION
VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION VALUE="P5">P5</SELECT>

</td>
<td align=left valign=top>
<SELECT NAME="bug_severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION
VALUE="major">major<OPTION VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION
VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```







Why Simplifying?

First of all: *simplified test cases can pinpoint failure causes.*

Besides that:

A simplified test case is easier to communicate. Is it relevant that the margins be set to .50? If the failure occurs nonetheless, we can leave away this detail.

A simplified test case facilitates debugging. A smaller HTML input leads to a simpler program state.

Simplified test cases identify duplicate problem reports. If we know that some specific HTML tag causes printing to fail, we can search for this HTML tag in other problem reports, marking them as duplicates.

The Gecko BugAThon

Idea—Volunteers help the Mozilla programmers by creating *simplified test cases*:

> 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.

Pledges	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*

Simplification Levels

Cutting away input can be done on three levels:

Lexical simplification. Cut away single characters or lines, without any consideration of the syntax or semantics of the input.

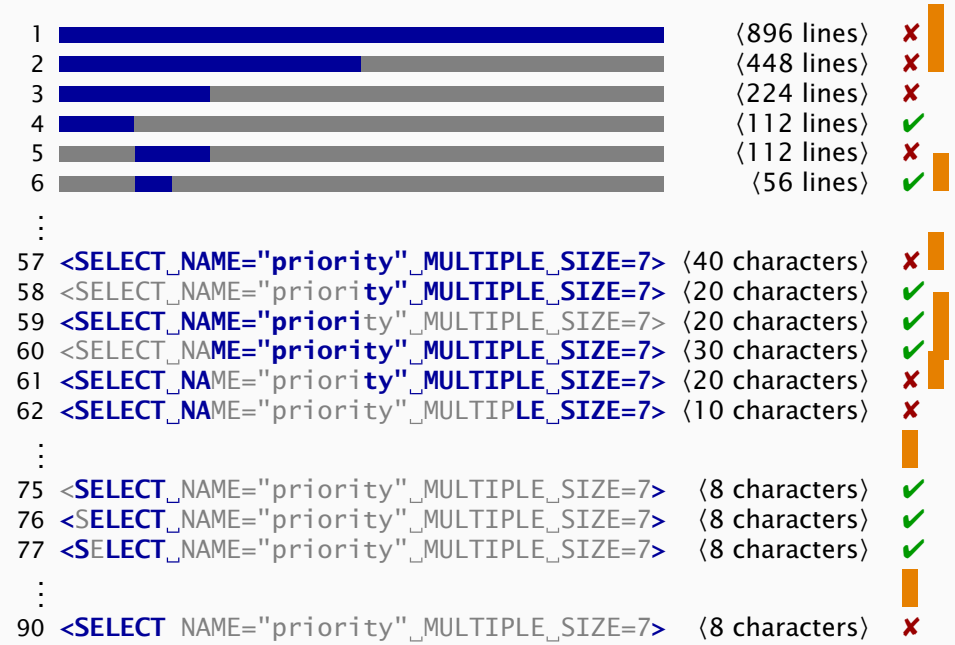
Syntactic simplification. Only cut away (HTML) *substructures*, avoiding syntax errors. In most cases, the best compromise.

Semantic simplification. Additionally, ensure *cross-references* (links and targets) are all valid.

Mozilla is not picky about syntax ⇒ lexical simplification!

Simplifying HTML Input

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



Simplified bug report: `Printing <SELECT> crashes.`

Delta Debugging

DELTA DEBUGGING: automates simplification of test cases

Basic idea:

- create subsets of failure-inducing circumstances
  (to be precise: subsets of failure-inducing *differences* between circumstances)
- test them automatically
- depending on the outcome,
  - further refine the subset or
  - try the next viable alternative.







Delta Debugging (2)

Like DIVIDE AND CONQUER, DELTA DEBUGGING is a very *general* concept.

DELTA DEBUGGING can be used to isolate

- *failure-inducing circumstances*, especially *program input* (see today and next lecture)
- *failure-inducing thread schedules*
- *failure-inducing code changes*
- *Failure-inducing program states* (later on)







Delta Debugging (3)

DELTA DEBUGGING has three building blocks:

An automated *test*, returning ✘ (“fail”) or ✔ (“pass”) depending on whether the failure occurs or not—or **?** (“unresolved”).

In our concrete example, such a test would generate HTML code and use CAPTURE AND REPLAY to instrument Mozilla.

A means to *decompose* the cause into smaller parts. In our concrete example, we search for the failure cause in the *input;* hence, we decompose the input.

A *strategy* on how and what to test. A simple, yet effective simplification strategy is the one shown before: Start by removing large chunks, increasing granularity as needed.

Delta Debugging (4)

DELTA DEBUGGING returns a *1-minimal* test case
⇒ every single part of the simplified test case is *relevant.*

> 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.

Complexity ranges from logarithmic (best case) to quadratic (worst case; pathologic)







Simplifying User Interaction

Let us recall the problem report:

> 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

Is setting the file name really relevant?

Do we have to set the margins to .50?







Simplifying User Interaction (2)

Basic idea:

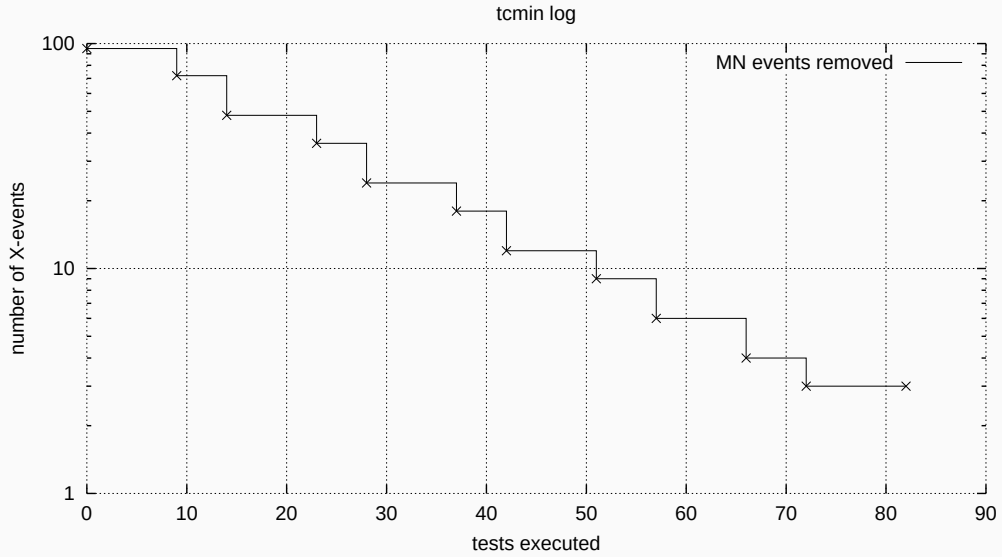
- Record the *user interaction* with CAPTURE AND REPLAY
- Set up an automated test that replays *subsequences* of the interaction
- Simplify the user interaction with DELTA DEBUGGING







Simplifying User Interaction (3)



Simplifying User Interaction (4)

Simplified User Interaction:

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.)

Consequence: selecting *Print to File*, altering the default file name, setting print margins is all *irrelevant*.

Simplifed bug report:

```
Printing <SELECT> crashes.
```

Alternate Circumstances

The simplification concept can be easily extended to arbitrary *circumstances* that determine a program run:

- If your program depends on a set of environment variables, use DELTA DEBUGGING to simplify the environment to the relevant settings.
- If your program reads in a file, you can use DELTA DEBUGGING to simplify the file to the relevant contents.
- If your program relies on network interaction, and you can set up CAPTURE AND REPLAY such that this network interaction is reproduced, you can easily use DELTA DEBUGGING to simplify failure-inducing network interaction.

Circumstances

Let us now give some *formal* definitions of Delta Debugging.

Let $\mathcal{R}$ denote the set of possible *configurations* of circumstances that determine a program run.

Each $r \in \mathcal{R}$ determines a specific program run (hence the name $\mathcal{R}$), just as every HTML input determines a specific Mozilla behavior.

In the same vein, each $r \in \mathcal{R}$ determines a specific test outcome.







Tests

Our *testing function* takes a configuration $r \in \mathcal{R}$ and determines the *test outcome*:

- 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 **?**).

Definition 1 (*rtest*) *The function* $rtest : \mathcal{R} \to \{✘, ✔, ?\}$ *determines for a program run* $r \in \mathcal{R}$ *whether some specific failure occurs (✘) or not (✔) or whether the test is unresolved (**?**).*

Differences between Runs

$r_{✔} \in \mathcal{R}$ and $r_{✘} \in \mathcal{R}$ stand for a passing run and the failing run, respectively.

$r_{✘}$ is the run whose circumstances are to be simplified.

Our aim: To find a minimal *difference* between $r_{✔}$ and $r_{✘}$—that is, the actual failure cause.

Definition 2 (Change) *A* change δ *is a mapping* $\delta : \mathcal{R} \to \mathcal{R}$. *The set of changes is* $C = \mathcal{R}^{\mathcal{R}}$. *The* relevant change *between two runs* $r_{✔}, r_{✘} \in \mathcal{R}$ *is a change* $\delta \in C$ *such that* $\delta(r_{✔}) = r_{✘}$.

Decomposing Differences

We assume that δ can be decomposed into smaller differences:

$$\delta = \delta_1 \circ \delta_2 \circ \cdots \circ \delta_n$$

Definition 3 (Composition of changes) *The* change composition $\circ : C \times C \to C$ *is defined as* $(\delta_i \circ \delta_j)(r) = \delta_i(\delta_j(r))$.

Identifying Runs by Differences

For convenience, we identify each test case by *the set of changes being applied to* $r_{✔}$.

That is, we define $c_{✔}$ as the empty set $c_{✔} = \varnothing$ which identifies the passing run $r_{✔}$ (no changes applied).

The set of all changes $c_{✘} = \{\delta_1, \delta_2, \ldots, \delta_n\}$ identifies the failing run $r_{✘} = (\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(r_{✔})$.







Testing by Differences

Test cases are related to program runs by means of the *test* function, which applies the respective change set to $r_{✔}$ and tests the resulting run.

Definition 4 (*test*) *The function test* : $2^{c_{✘}} \to \{✘, ✔, ?\}$ *is defined as follows: Let* $c \subseteq c_{✘}$ *be a test case with* $c = \{\delta_1, \delta_2, \ldots, \delta_n\}$. *Then,* $test(c) = rtest((\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(r_{✔}))$ *holds.*

$test(c_{✔}) = rtest(r_{✔}) = ✔$ and
$test(c_{✘}) = rtest(\delta(r_{✔})) = rtest(r_{✘}) = ✘$ hold.

Simplified Runs

Our issue: simplifying $c_{\text{✘}}$ to a minimum.

First attempt—a global minimum:

Definition 5 (Global minimum) *A set $c \subseteq c_{\text{✘}}$ is called the* global minimum *of $c_{\text{✘}}$ if $\forall c' \subseteq c_{\text{✘}} \cdot (|c'| < |c| \Rightarrow \mathit{test}(c') \neq \text{✘})$ holds.*

Unfortunately, this requires testing all $2^{|c_{\text{✘}}|}$ subsets of $c_{\text{✘}}$.







Simplified Runs (2)

Second attempt—each part is relevant on its own:

Definition 6 (1-minimal test case) *A test case* $c \subseteq c_{\text{✘}}$ *is* 1-minimal *if* $\forall \delta_i \in c \cdot \mathit{test}(c \setminus \{\delta_i\}) \neq$ ✘ *holds.*

Generalization: n-minimality—any removal of at most n changes makes the failure disappear.

Linear Simplification

Idea: remove one change at a time while failure persists.

Example: failure occurs if changes 5 and 7 are applied







Step	Test case								*test*
1	.	2	3	4	5	6	7	8	✘
2	.	.	3	4	5	6	7	8	✘
3	.	.	.	4	5	6	7	8	✘
4	.	.	.	.	5	6	7	8	✘
5	.	.	.	.	.	6	7	8	✔
6	.	.	.	.	5	.	7	8	✘
7	.	.	.	.	5	.	.	8	✔
8	.	.	.	.	5	.	7	.	✘
9	.	.	.	.	.	.	7	.	✔
10	.	.	.	.	5	.	.	.	✔
Result	.	.	.	.	5	.	7	.	

Linear Simplification

Formal definition:

$$ddlin(c_{\text{✘}}) = ddlin'(c_{\text{✘}}) \quad \text{where}$$

$$ddlin'(c'_{\text{✘}}) = \begin{cases} ddlin'(c'_{\text{✘}} \setminus \{\delta_i\}) & \text{if } \exists \delta_i \in c'_{\text{✘}} \cdot test(c'_{\text{✘}} \setminus \{\delta_i\}) = \text{✘} \\ c'_{\text{✘}} & \text{otherwise} \end{cases}$$

Not effective enough—requires at least $|c_{\text{✘}}|$ tests!

Divide-and-Conquer Simplification

Basic idea: split changes into two halves;
if failure does not occur in either part, increase granularity

Step	Test case								*test*	
1	.	.	.	.	5	6	7	8	✘	
2	.	.	.	.	.	.	7	8	✔	
3	.	.	.	.	5	6	.	.	✔	Increase $n = 4$
4	.	.	.	.	.	6	7	8	✔	
5	.	.	.	.	5	.	7	8	✘	Restart with $n = 2$
6	.	.	.	.	.	.	.	8	✔	
7	.	.	.	.	5	.	7	.	✘	
8	.	.	.	.	.	.	7	.	✔	
9	.	.	.	.	5	.	.	.	✔	
Result	.	.	.	.	5	.	7	.		

Divide-and-Conquer Simplification (2)

Formal definition:

$$ddmin(c_{\text{✘}}) = ddmin'(c_{\text{✘}}, 2) \quad \text{where}$$

$$ddmin'(c'_{\text{✘}}, n) = \begin{cases} ddmin'(\nabla_i, \max(n-1, 2)) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \text{✘} \\ ddmin'(c'_{\text{✘}}, \min(2n, |c_{\text{✘}}|)) & \text{if } 2n < |c_{\text{✘}}| \\ c'_{\text{✘}} & \text{otherwise} \end{cases}$$

with $c'_{\text{✘}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n, \nabla_i = c'_{\text{✘}} \setminus \Delta_i$, and

$\forall \Delta_i, \Delta_j \cdot \Delta_i \cap \Delta_j = \varnothing \wedge |\Delta_i| \approx |\Delta_j|$.

Divide-and-Conquer Simplification (3)

Basic properties of *ddmin*:

Worst-case complexity is quadratic. Pathological example: every complement is unresolved—except for the last.

Best-case complexity is logarithmic. If a small part of the input induces the failure (like <SELECT> in the HTML input), we're set.

Important issue, though: We must avoid unresolved test outcomes!







Simplifying GCC Input

```
#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);
}
```

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

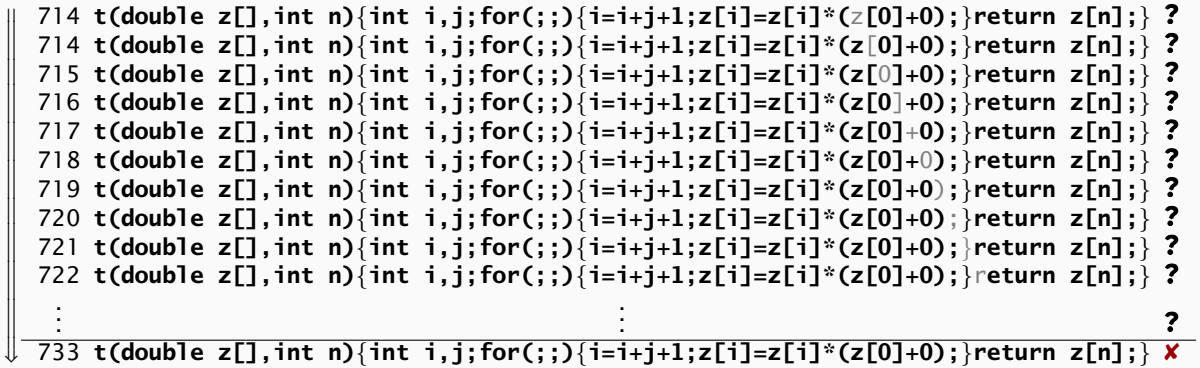
Simplifying GCC Input (2)

Step	Configuration	*test*
1	#define SIZE 20 ... double **mult**(...) { ... }	✘
2	#define SIZE 20	✔
3	double **mult**(...) { ... }	✘
4	double **mult**(...) { int i, j; i = 0; }	✔
5	double **mult**(...) { for(...) { ... } ... }	**?**
⋮	⋮	⋮

Minimal input found after 857 tests:

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

The Curse of Lexical Simplification



Syntactical Simplification

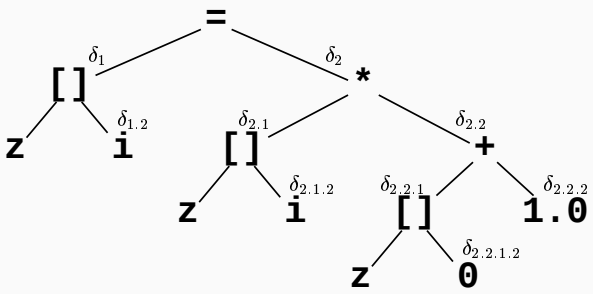
Basic idea: Turn input into a *syntax tree* and replace subtrees by (trivial) replacements

Example—Input `z[i] = z[i] * (z[0] + 1.0)`








=
δ_1
δ_2
[]
*
z
$\delta_{1.2}$
i
$\delta_{2.1}$
$\delta_{2.2}$
[]
+
z
$\delta_{2.1.2}$
i
$\delta_{2.2.1}$
$\delta_{2.2.2}$
[]
1.0
z
$\delta_{2.2.1.2}$
0

Syntactical Simplification (2)

Step	Applied deltas	Input	Test
$r_{\times}$	1 1.2 2 2.1 2.1.2 2.2 2.2.1 2.2.1.2 2.2.2	`z[i] = z[i] * (z[0] + 1.0)`	✘
$r_{\checkmark}$	. . . .	⟨empty⟩	✔
Granularity $n = 2$			
1	. . 2 2.1 2.1.2 2.2 2.2.1 2.2.1.2 2.2.2	`z = z[i] * (z[0] + 1.0)`	✔
2	1 1.2	`z[i] = 0`	✔
Granularity $n = 4$			
3	. 1.2 2 2.1 2.1.2 2.2 2.2.1 2.2.1.2 2.2.2	⟨infeasible⟩	?
4	1 . 2 2.1 2.1.2 2.2 2.2.1 2.2.1.2 2.2.2	`z[0] = z[i] * (z[0] + 1.0)`	✔
5	1 1.2 . . . 2.2 2.2.1 2.2.1.2 2.2.2	⟨infeasible⟩	?
6	1 1.2 2 2.1 2.1.2	`z[i] = z[i] * 0`	✔
Granularity $n = 8$			
7	. 1.2 2 2.1 2.1.2 2.2 2.2.1 2.2.1.2 2.2.2	⟨infeasible⟩	?
8	1 . 2 2.1 2.1.2 2.2 2.2.1 2.2.1.2 2.2.2	`z[0] = z[i] * (z[0] + 1.0)`	✔
9	1 1.2 . 2.1 2.1.2 2.2 2.2.1 2.2.1.2 2.2.2	⟨infeasible⟩	?
10	1 1.2 2 . 2.1.2 2.2 2.2.1 2.2.1.2 2.2.2	⟨infeasible⟩	?
11	1 1.2 2 2.1 . 2.2 2.2.1 2.2.1.2 2.2.2	`z[0] = z[0] * (z[0] + 1.0)`	✔
12	1 1.2 2 2.1 2.1.2 . 2.2.1 2.2.1.2 2.2.2	⟨infeasible⟩	?
13	1 1.2 2 2.1 2.1.2 2.2 . . 2.2.2	`z[0] = z[i] * (0 + 1.0)`	?
14	1 1.2 2 2.1 2.1.2 2.2 2.2.1 2.2.1.2 .	`z[0] = z[i] * (z[0] + 0)`	✔







Is it "the" Failure?

Problem: Our automated test must check whether the occurring failure is the same as the original failure.

As an analogon, consider this example:

> Billy and Suzy throw rocks at a bottle. Suzy throws first so that her rock arrives first and shatters the glass. Without Suzy's throw, Billy's throw would have shattered the bottle.

What's the cause of the bottle being shattered?

Is it “the” Failure? (2)

If Suzy had *not* thrown the rock, Billy’s throw would have shattered the bottle anyway.

Following the definition of “cause”, Suzy’s throw would not be the cause of the bottle being shattered, which is in contrast to common sense.







Is it "the" Failure? (3)

Solution: take into account not only *whether* the effect occurs, but also *when* and *how*.

Since Billy's rock would have smashed the bottle at another time—and probably, in some other way, too—, the effect would have been *different*.

Hence, Suzy's throw may not be the cause for the bottle being shattered in general—but the throw is certainly the cause for the bottle having been shattered just as we find it.







Is it “the” Failure? (4)

To compare the *when* and *how* of a failure, use a *backtrace*—the current program counter and the stack of calling functions at the moment of the failure.

When a failure occurs, the automated test can compare its backtrace with the backtrace of the original failure.

Only if both are equal must the automated test return ✘; otherwise, a different failure has occurred, and the automated test must return **?**.







Concepts

- DIVIDE AND CONQUER is useful to invent and modify hypotheses about failure causes. Its main idea is to break some initial (trivial) cause into smaller parts and to narrow down the actual cause along the scientific method.
- Simplified test cases play an important role in debugging, as they may pinpoint failure causes. They also make test cases easier to communicate, they facilitate debugging, and they identify duplicate problem reports.
- Simplification can be done on a lexical, on a syntactical, and on a semantical level.

Concepts (2)

- DELTA DEBUGGING automates simplification of test cases by systematically narrowing the difference between a passing and a failing run.
- The result of a DELTA DEBUGGING run is a *1-minimal* test case in which every part is relevant for producing the failure.
- DELTA DEBUGGING can be applied to arbitrary circumstances besides program input, as long as they can be captured and altered.







Concepts (3)

- The *ddmin* algorithm realizes a simple DIVIDE AND CONQUER strategy for Delta Debugging.
- It has quadratic complexity in the worst (pathological) case, and logarithmic complexity in the best case.







Concepts (4)

- The more knowledge about syntax and semantics of the input goes into decomposition, the more efficient is the simplification process.
- Syntactic simplification works on *syntax trees* rather than on streams of characters; it avoids unresolved test outcomes due to syntax errors.
- When checking a program run, be sure to compare the *when* and *how* of the failure with the properties of the original failure. This is best done by comparing the *backtrace*.