13th Int'l Workshop on the Principles of Diagnosis (DX-2002), Semmering, Austria, May 4th, 2002

Why does my program fail?

Isolating failure causes automatically

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Diagnostics under Total Control

In classical diagnosis settings,

- *measuring is expensive* (hence simulation and models)
- *experimentation is expensive* (hence diagnosis)

Diagnostics under Total Control

In classical diagnosis settings,

- *measuring is expensive* (hence simulation and models)
- *experimentation is expensive* (hence diagnosis)

But what if...

- ...the subject of diagnosis can be *arbitrarily examined*?
- ...experimentation is *cheap* and automatic?
- ...we can actually *verify* whether a cause is a cause?

Welcome to the world of automated debugging!

A True Story

Mozilla: Netscape's open source web browser

Developed by dozens of Netscape engineers and zillions of volunteers

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
```

Why does Mozilla crash?

We want to determine the *cause* of the Mozilla crash:

> *The **cause** of any event (“**effect**”) is a preceding event without which the effect would not have occurred.*
>
> — Microsoft Encarta

Why does Mozilla crash?

We want to determine the *cause* of the Mozilla crash:

> *The **cause** of any event ("**effect**") is a preceding event without which the effect would not have occurred.*
>
> — Microsoft Encarta

To prove causality, we must show experimentally that

1. the effect occurs when the cause occurs
2. the effect does not occur when the cause does not occur

In our case, the *effect* is Mozilla crashing.
The *cause* must be something different – e.g. the HTML input.

Our Issue: Simple Causes

A cause alone does not suffice – the cause must be *simple*, too:

- Simple test case ⇒ *simple program state*
- Simple test case ⇒ *general representative*

Mozilla BugAThon – Volunteers *simplify test cases*:

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*

Can't we automate this?

Simplifying HTML Input

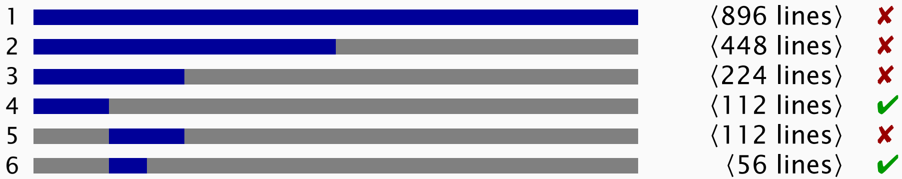
Basic idea: We use an *automated test* to simplify HTML pages, until each character is *relevant for the failure*:

Simplifying HTML Input

Basic idea: We use an *automated test* to simplify HTML pages, until each character is *relevant for the failure*:

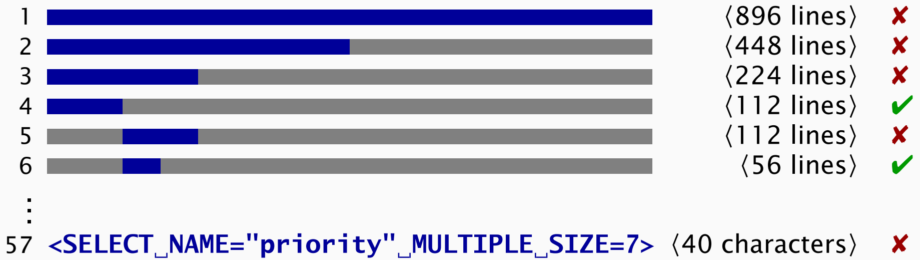
Simplifying HTML Input

Basic idea: We use an *automated test* to simplify HTML pages, until each character is *relevant for the failure*:



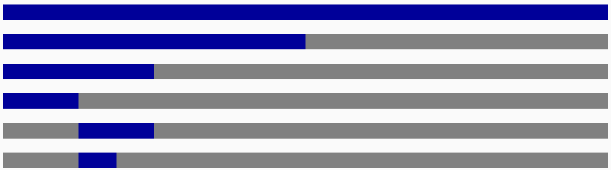
Simplifying HTML Input

Basic idea: We use an *automated test* to simplify HTML pages, until each character is *relevant for the failure*:



Simplifying HTML Input

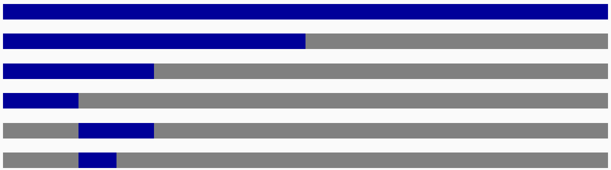
Basic idea: We use an *automated test* to simplify HTML pages, until each character is *relevant for the failure*:



1		⟨896 lines⟩	✘
2		⟨448 lines⟩	✘
3		⟨224 lines⟩	✘
4		⟨112 lines⟩	✔
5		⟨112 lines⟩	✘
6		⟨56 lines⟩	✔
⋮			
57	**<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>**	⟨40 characters⟩	✘
58	<SELECT␣NAME="priori**ty"␣MULTIPLE␣SIZE=7>**	⟨20 characters⟩	✔
59	**<SELECT␣NAME="priori**ty"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔

Simplifying HTML Input

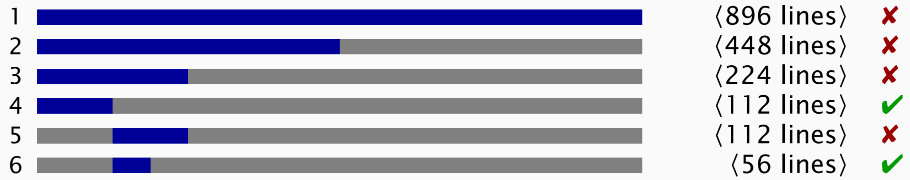
Basic idea: We use an *automated test* to simplify HTML pages, until each character is *relevant for the failure*:



1		⟨896 lines⟩	✘
2		⟨448 lines⟩	✘
3		⟨224 lines⟩	✘
4		⟨112 lines⟩	✔
5		⟨112 lines⟩	✘
6		⟨56 lines⟩	✔
⋮			
57	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨40 characters⟩	✘
58	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
59	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
60	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨30 characters⟩	✔

Simplifying HTML Input

Basic idea: We use an *automated test* to simplify HTML pages, until each character is *relevant for the failure*:

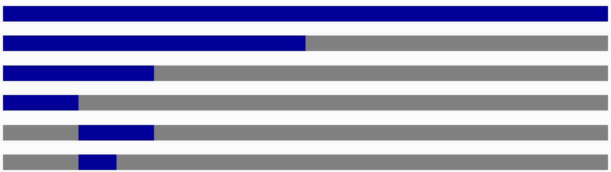


⋮

57	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨40 characters⟩	✘
58	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
59	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
60	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨30 characters⟩	✔
61	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✘

Simplifying HTML Input

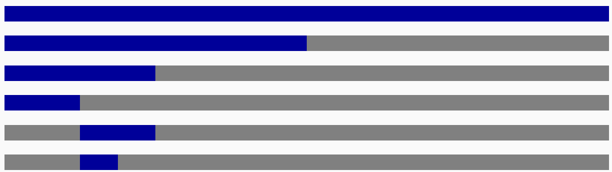
Basic idea: We use an *automated test* to simplify HTML pages, until each character is *relevant for the failure*:



1		⟨896 lines⟩	✘
2		⟨448 lines⟩	✘
3		⟨224 lines⟩	✘
4		⟨112 lines⟩	✔
5		⟨112 lines⟩	✘
6		⟨56 lines⟩	✔
⋮			
57	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨40 characters⟩	✘
58	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
59	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
60	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨30 characters⟩	✔
61	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✘
62	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨10 characters⟩	✘
⋮			

Simplifying HTML Input

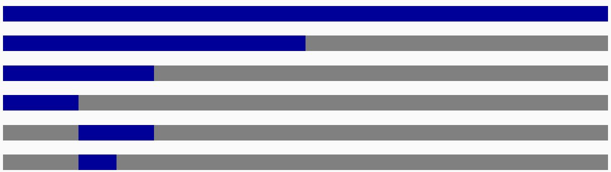
Basic idea: We use an *automated test* to simplify HTML pages, until each character is *relevant for the failure*:



1		⟨896 lines⟩	✘
2		⟨448 lines⟩	✘
3		⟨224 lines⟩	✘
4		⟨112 lines⟩	✔
5		⟨112 lines⟩	✘
6		⟨56 lines⟩	✔
⋮			
57	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨40 characters⟩	✘
58	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
59	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
60	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨30 characters⟩	✔
61	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✘
62	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨10 characters⟩	✘
⋮			
75	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
76	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
77	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
⋮			

Simplifying HTML Input

Basic idea: We use an *automated test* to simplify HTML pages, until each character is *relevant for the failure*:



1		⟨896 lines⟩	✘
2		⟨448 lines⟩	✘
3		⟨224 lines⟩	✘
4		⟨112 lines⟩	✔
5		⟨112 lines⟩	✘
6		⟨56 lines⟩	✔
⋮			
57	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨40 characters⟩	✘
58	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
59	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
60	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨30 characters⟩	✔
61	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✘
62	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨10 characters⟩	✘
⋮			
75	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
76	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
77	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
⋮			
90	<SELECT NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✘

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

The Delta Debugging Algorithm

Given: $test, c_{✔}, c_{✘} \cdot c_{✔} \subseteq c_{✘} \land test(c_{✔}) = ✔ \land test(c_{✘}) = ✘$.

The Delta Debugging Algorithm

Given: $test, c_{\checkmark}, c_{\text{✘}} \cdot c_{\checkmark} \subseteq c_{\text{✘}} \wedge test(c_{\checkmark}) = \checkmark \wedge test(c_{\text{✘}}) = \text{✘}$.

Goal: $c'_{\text{✘}} = ddmin(c_{\text{✘}})$ such that $c_{\checkmark} \subseteq c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✘}}) = \text{✘}$ and each element of $c'_{\text{✘}}$ is *relevant for the failure*.

The Delta Debugging Algorithm

Given: $test, c_{✔}, c_{✘} \cdot c_{✔} \subseteq c_{✘} \wedge test(c_{✔}) = ✔ \wedge test(c_{✘}) = ✘$.

Goal: $c'_{✘} = ddmin(c_{✘})$ such that $c_{✔} \subseteq c'_{✘} \subseteq c_{✘}$, $test(c'_{✘}) = ✘$ and each element of $c'_{✘}$ is *relevant for the failure*.

Let $\Delta = c'_{✘} - c_{✔} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$

The Delta Debugging Algorithm

Given: $test, c_{\checkmark}, c_{\times} \cdot c_{\checkmark} \subseteq c_{\times} \wedge test(c_{\checkmark}) = \checkmark \wedge test(c_{\times}) = \times$.

Goal: $c'_{\times} = ddmin(c_{\times})$ such that $c_{\checkmark} \subseteq c'_{\times} \subseteq c_{\times}$, $test(c'_{\times}) = \times$ and each element of $c'_{\times}$ is *relevant for the failure*.

Let $\Delta = c'_{\times} - c_{\checkmark} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$ in

$$ddmin(c_{\times}) = ddmin_2(c_{\times}, 2) \text{ where } ddmin_2(c'_{\times}, n) =$$

$$\begin{cases} ddmin_2(c_{\checkmark} \cup \Delta_i, 2) & \text{if } \exists i \cdot test(c_{\checkmark} \cup \Delta_i) = \times \\ & \\ & \end{cases}$$

The Delta Debugging Algorithm

Given: $test, c_{✔}, c_{✘} \cdot c_{✔} \subseteq c_{✘} \wedge test(c_{✔}) = ✔ \wedge test(c_{✘}) = ✘$.

Goal: $c'_{✘} = ddmin(c_{✘})$ such that $c_{✔} \subseteq c'_{✘} \subseteq c_{✘}$, $test(c'_{✘}) = ✘$ and each element of $c'_{✘}$ is *relevant for the failure*.

Let $\Delta = c'_{✘} - c_{✔} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$ in

$$ddmin(c_{✘}) = ddmin_2(c_{✘}, 2) \text{ where } ddmin_2(c'_{✘}, n) =$$

$$\begin{cases} ddmin_2(c_{✔} \cup \Delta_i, 2) & \text{if } \exists i \cdot test(c_{✔} \cup \Delta_i) = ✘ \\ ddmin_2(c'_{✘} - \Delta_i, \max(n-1, 2)) & \text{if } \exists i \cdot test(c'_{✘} - \Delta_i) = ✘ \\ & \end{cases}$$

The Delta Debugging Algorithm

Given: $test, c_{✔}, c_{✘} \cdot c_{✔} \subseteq c_{✘} \wedge test(c_{✔}) = ✔ \wedge test(c_{✘}) = ✘$.

Goal: $c'_{✘} = ddmin(c_{✘})$ such that $c_{✔} \subseteq c'_{✘} \subseteq c_{✘}$, $test(c'_{✘}) = ✘$ and each element of $c'_{✘}$ is *relevant for the failure*.

Let $\Delta = c'_{✘} - c_{✔} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$ in

$$ddmin(c_{✘}) = ddmin_2(c_{✘}, 2) \text{ where } ddmin_2(c'_{✘}, n) =$$

$$\begin{cases} ddmin_2(c_{✔} \cup \Delta_i, 2) & \text{if } \exists i \cdot test(c_{✔} \cup \Delta_i) = ✘ \\ ddmin_2(c'_{✘} - \Delta_i, \max(n-1, 2)) & \text{if } \exists i \cdot test(c'_{✘} - \Delta_i) = ✘ \\ ddmin_2(c'_{✘}, \min(2n, |\Delta|)) & \text{if } n < |\Delta| \\ c'_{✘} & \text{otherwise} \end{cases}$$

The Delta Debugging Algorithm

Given: $test, c_{\checkmark}, c_{\times} \cdot c_{\checkmark} \subseteq c_{\times} \wedge test(c_{\checkmark}) = \checkmark \wedge test(c_{\times}) = \times$.

Goal: $c'_{\times} = ddmin(c_{\times})$ such that $c_{\checkmark} \subseteq c'_{\times} \subseteq c_{\times}$, $test(c'_{\times}) = \times$ and each element of $c'_{\times}$ is *relevant for the failure*.

Let $\Delta = c'_{\times} - c_{\checkmark} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$ in

$$ddmin(c_{\times}) = ddmin_2(c_{\times}, 2) \text{ where } ddmin_2(c'_{\times}, n) =$$

$$\begin{cases} ddmin_2(c_{\checkmark} \cup \Delta_i, 2) & \text{if } \exists i \cdot test(c_{\checkmark} \cup \Delta_i) = \times \\ ddmin_2(c'_{\times} - \Delta_i, \max(n-1, 2)) & \text{if } \exists i \cdot test(c'_{\times} - \Delta_i) = \times \\ ddmin_2(c'_{\times}, \min(2n, |\Delta|)) & \text{if } n < |\Delta| \\ c'_{\times} & \text{otherwise} \end{cases}$$

Number of tests: between $2\log_2(|c_{\times} - c_{\checkmark}|)$ and $|c_{\times} - c_{\checkmark}|^2$; no less than $2|c'_{\times}|$.

Δ: *arbitrary circumstances* (input, code changes, threads...)

GCC eats up Desktop

Simplifying *complex inputs* can be expensive.

```
double bug(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] + 0.0);
  }
  return z[n];
}
```

`bug.c` causes the GNU compiler (GCC 2.95.2) to crash:

```
linux$ gcc -O bug.c
gcc: Internal error: program cc1 got fatal signal 11
linux$ _
```

Simplifying GCC Input

Once again, we simplify the GCC input `bug.c`.

Problem: Even if we take the C syntax into account, there are still *unresolved test outcomes* (**?**)

#	GCC input	test
1	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✘
2	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	?
3	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔
4	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✘
5	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	?
6	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✘
7	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔
⋮		

Determining the simplified input requires *857 tests*:

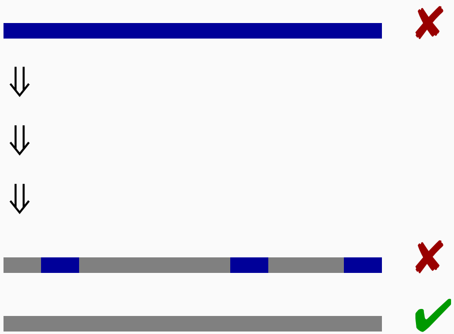
```
g(double z[],int n){int i,j;for(;;){i = i + j + 1;z[i] = z[i] * (z[0] + 0);}return z[n];}
```

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

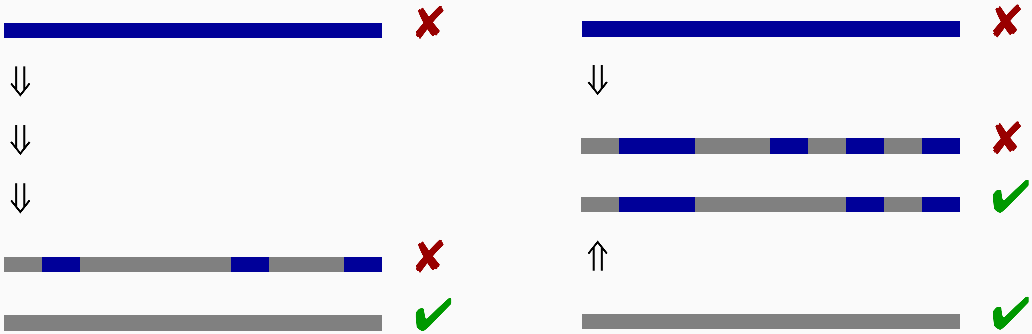


Simplifying vs. Isolating

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*.

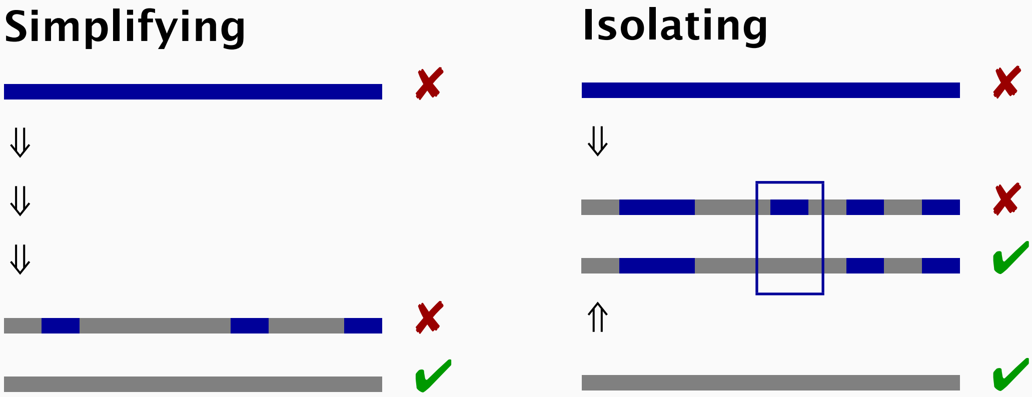
Simplifying | **Isolating**



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*.



Larger context – but fewer tests and smaller causes

Isolating Failure Causes

We isolate the failure-inducing GCC input:

#	GCC input	test
1	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✘
2	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✔

Isolating Failure Causes

We isolate the failure-inducing GCC input:

#	GCC input	test
1	double **bug**(. . .) { int i, j; i = 0; for (. . .) { . . . } . . . }	✘
3	double **bug**(. . .) { int i, j; i = 0; for (. . .) { . . . } . . . }	
2	double **bug**(. . .) { int i, j; i = 0; for (. . .) { . . . } . . . }	✔

Isolating Failure Causes

We isolate the failure-inducing GCC input:

#	GCC input	test
1	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✘
3	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✔
2	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✔

Isolating Failure Causes

We isolate the failure-inducing GCC input:

#	GCC input	test
1	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✘
4	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	
3	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔
2	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔

Isolating Failure Causes

We isolate the failure-inducing GCC input:

#	GCC input	test
1	double **bug**(. . .) { int i, j; i = 0; for (. . .) { . . . } . . . }	✘
4	double **bug**(. . .) { int i, j; i = 0; for (. . .) { . . . } . . . }	✔
3	double **bug**(. . .) { int i, j; i = 0; for (. . .) { . . . } . . . }	✔
2	double **bug**(. . .) { int i, j; i = 0; for (. . .) { . . . } . . . }	✔

Isolating Failure Causes

We isolate the failure-inducing GCC input:

#	GCC input	test
1	double **bug**(...) { int i, j; i = 0; for (...) { ... } ... }	✘
5	double **bug**(...) { int i, j; i = 0; for (...) { ... } ... }	
4	double **bug**(...) { int i, j; i = 0; for (...) { ... } ... }	✔
3	double **bug**(...) { int i, j; i = 0; for (...) { ... } ... }	✔
2	double **bug**(...) { int i, j; i = 0; for (...) { ... } ... }	✔

Isolating Failure Causes

We isolate the failure-inducing GCC input:

#	GCC input	test
1	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✘
5	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✘
4	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔
3	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔
2	double **bug**(...) { int *i*, *j*; *i* = 0; for (...) { ... } ... }	✔

Isolating Failure Causes

We isolate the failure-inducing GCC input:

#	GCC input	test
1	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✘
5	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✘
⋮		
19	... $z[i] = z[i] * (z[0] + 0.0)$; ...	✘
18	... $z[i] = z[i] * (z[0] + 0.0)$; ...	✔
⋮		
4	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✔
3	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✔
2	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✔

$+\ 0.0$ is the failure cause – after only 19 tests ($\approx$ 2 seconds)

Isolating Failure Causes

We isolate the failure-inducing GCC input:

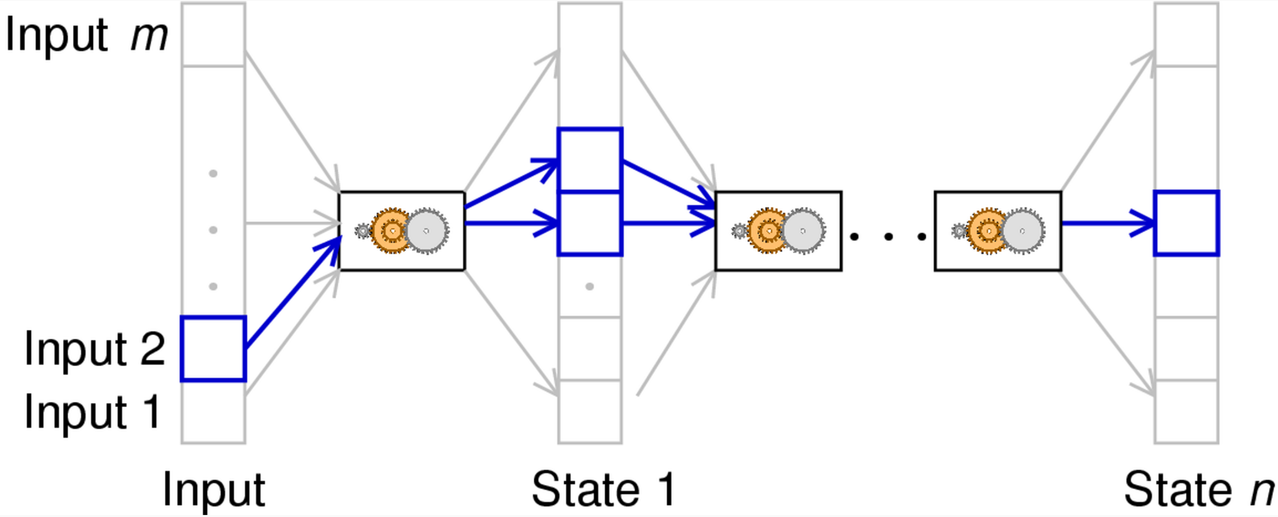
#	GCC input	test
1	double **bug**(...) { int i, j; i = 0; for (...) { ... } ... }	✘
5	double **bug**(...) { int i, j; i = 0; for (...) { ... } ... }	✘
⋮		
19	... $z[i] = z[i] * (z[0] + 0.0);$ ...	✘
18	... $z[i] = z[i] * (z[0] + 0.0);$ ...	✔
⋮		
4	double **bug**(...) { int i, j; i = 0; for (...) { ... } ... }	✔
3	double **bug**(...) { int i, j; i = 0; for (...) { ... } ... }	✔
2	double **bug**(...) { int i, j; i = 0; for (...) { ... } ... }	✔

$+ 0.0$ is the failure cause – after only 19 tests ($\approx$ 2 seconds)

(Compare this to, say, 1 man-hour to isolate bad PostScript code :–)

What's going on in GCC?

The difference $+0.0$ is just the beginning of a *cause-effect chain* within the GCC run.



Each difference induces later state changes.
But only some of these effects are relevant for the failure.
Goal: *Isolate failure-inducing state changes!*

Deltas between States

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

Example: GCC state in the function *combine_instructions*

#	reg_rtx_no	cur_insn_uid	last_linenum	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✘

Deltas between States

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

Example: GCC state in the function *combine_instructions*

#	reg_rtx_no	cur_insn_uid	last_linenum	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✘
2	31	70	14	0x81fc4a0	✔

Deltas between States

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

Example: GCC state in the function *combine_instructions*

#	reg_rtx_no	cur_insn_uid	last_linenum	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✘
3	32	74	14	0x81fc4a0	
2	31	70	14	0x81fc4a0	✔

Deltas between States

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

Example: GCC state in the function *combine_instructions*

#	reg_rtx_no	cur_insn_uid	last_linenum	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✘
3	32	74	14	0x81fc4a0	✔
2	31	70	14	0x81fc4a0	✔

Deltas between States

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

Example: GCC state in the function *combine_instructions*

#	reg_rtx_no	cur_insn_uid	last_linenum	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✘
4	32	74	14	0x81fc4e4	
3	32	74	14	0x81fc4a0	✔
2	31	70	14	0x81fc4a0	✔

Deltas between States

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

Example: GCC state in the function *combine_instructions*

#	reg_rtx_no	cur_insn_uid	last_linenum	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✘
4	32	74	14	0x81fc4e4	?
3	32	74	14	0x81fc4a0	✔
2	31	70	14	0x81fc4a0	✔

Deltas between States

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

Example: GCC state in the function *combine_instructions*

#	reg_rtx_no	cur_insn_uid	last_linenum	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✘
5	32	74	15	0x81fc4a0	✔
4	32	74	14	0x81fc4e4	?
3	32	74	14	0x81fc4a0	✔
2	31	70	14	0x81fc4a0	✔

Deltas between States

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

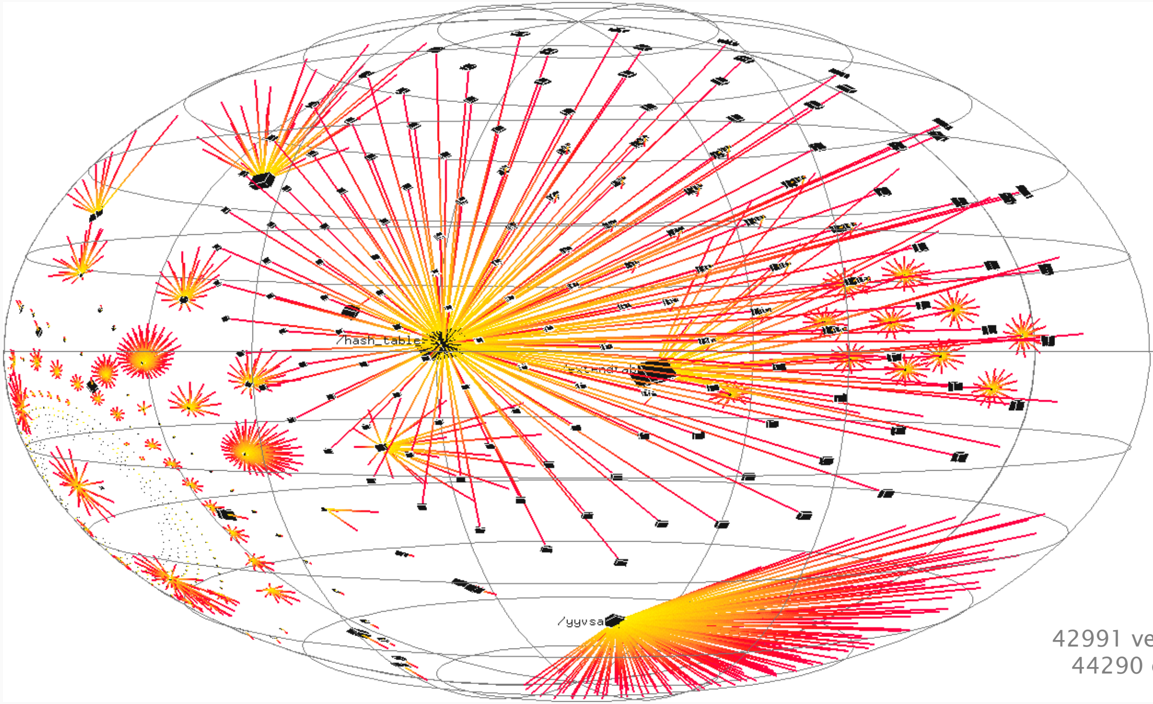
Example: GCC state in the function *combine_instructions*

#	reg_rtx_no	cur_insn_uid	last_linenum	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✘
			?		
5	32	74	15	0x81fc4a0	✔
4	32	74	14	0x81fc4e4	?
3	32	74	14	0x81fc4a0	✔
2	31	70	14	0x81fc4a0	✔

Life is not so simple – we must also determine *structural differences* and apply them!

The GCC Memory Graph

We extract the program state as *graph*:
Vertices are *variables*, edges are *references*

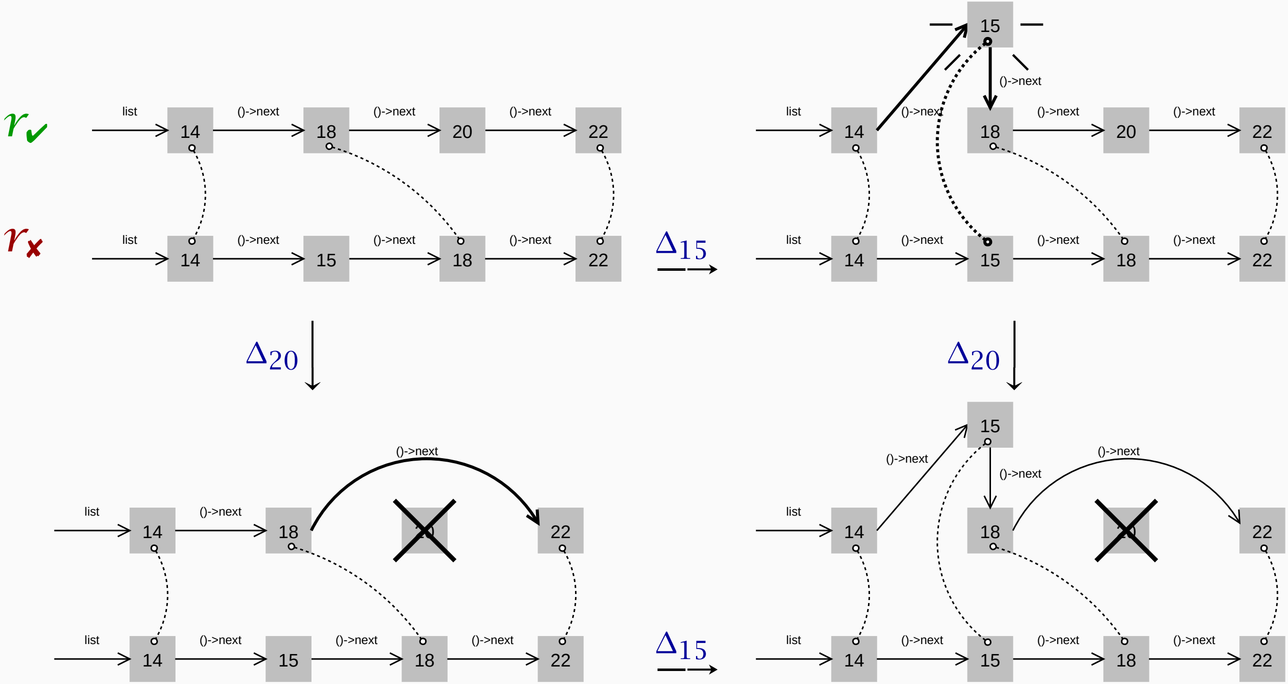


42991 vertices
44290 edges

Structural Differences

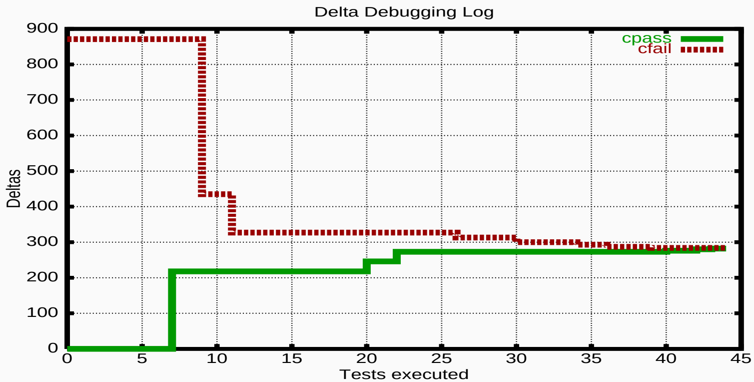
On graphs, one can compute structural differences:

Δ_{15} creates a variable, Δ_{20} deletes one



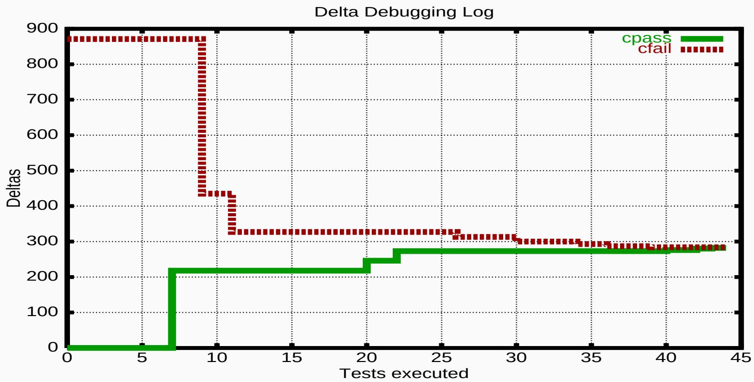
Relevant State Differences

Our prototype HOWCOME examines the state of cc1 in *combine_instructions*: 871 nodes (= variables) are different



Relevant State Differences

Our prototype HOWCOME examines the state of cc1 in *combine_instructions*: 871 nodes (= variables) are different



Only one variable causes the failure:

```
$m = (struct rtx_def *)malloc(12)
$m->code = PLUS
first_loop_store_insn->fld[1]...rtx = $m
```

The GCC Cause-Effect Chain

After 59 tests, HOWCOME has determined these failure causes:

```
Cause-effect chain for './gcc/cc1'
Arguments are '-O' 'bug.i' (instead of '-O' 'ok.i')
therefore at 'main',
  argv[2] = "bug.i"
  (instead of "ok.i")
```

The GCC Cause-Effect Chain

After 59 tests, HOWCOME has determined these failure causes:

```
Cause-effect chain for './gcc/cc1'
Arguments are '-O' 'bug.i' (instead of '-O' 'ok.i')
therefore at 'main',
  argv[2] = "bug.i"
  (instead of "ok.i")
therefore at 'combine_instructions',
  *first_loop_store_insn->fld[1].rtx->
  fld[1].rtx->fld[3].rtx->fld[1].rtx = ⟨new variable⟩
```

The GCC Cause-Effect Chain

After 59 tests, HOWCOME has determined these failure causes:

```
Cause-effect chain for './gcc/cc1'
Arguments are '-O' 'bug.i' (instead of '-O' 'ok.i')
therefore at 'main',
  argv[2] = "bug.i"
  (instead of "ok.i")
therefore at 'combine_instructions',
  *first_loop_store_insn->fld[1].rtx->
  fld[1].rtx->fld[3].rtx->fld[1].rtx = ⟨new variable⟩
therefore at 'if_then_else_cond',
  link->fld[0].rtx->fld[0].rtx = &link
  (instead of i1dest)
therefore the run fails.
```

Total running time: 6 seconds

The GCC Cause-Effect Chain

After 59 tests, HOWCOME has determined these failure causes:

```
Cause-effect chain for './gcc/cc1'
Arguments are '-O' 'bug.i' (instead of '-O' 'ok.i')
therefore at 'main',
  argv[2] = "bug.i"
  (instead of "ok.i")
therefore at 'combine_instructions',
  *first_loop_store_insn->fld[1].rtx->
  fld[1].rtx->fld[3].rtx->fld[1].rtx = ⟨new variable⟩
therefore at 'if_then_else_cond',
  link->fld[0].rtx->fld[0].rtx = &link
  (instead of i1dest)
therefore the run fails.
```

Total running time: 6 seconds

(+ 90 minutes for extracting the memory graph through GDB)

Perspectives

Program understanding is still in its infancy:

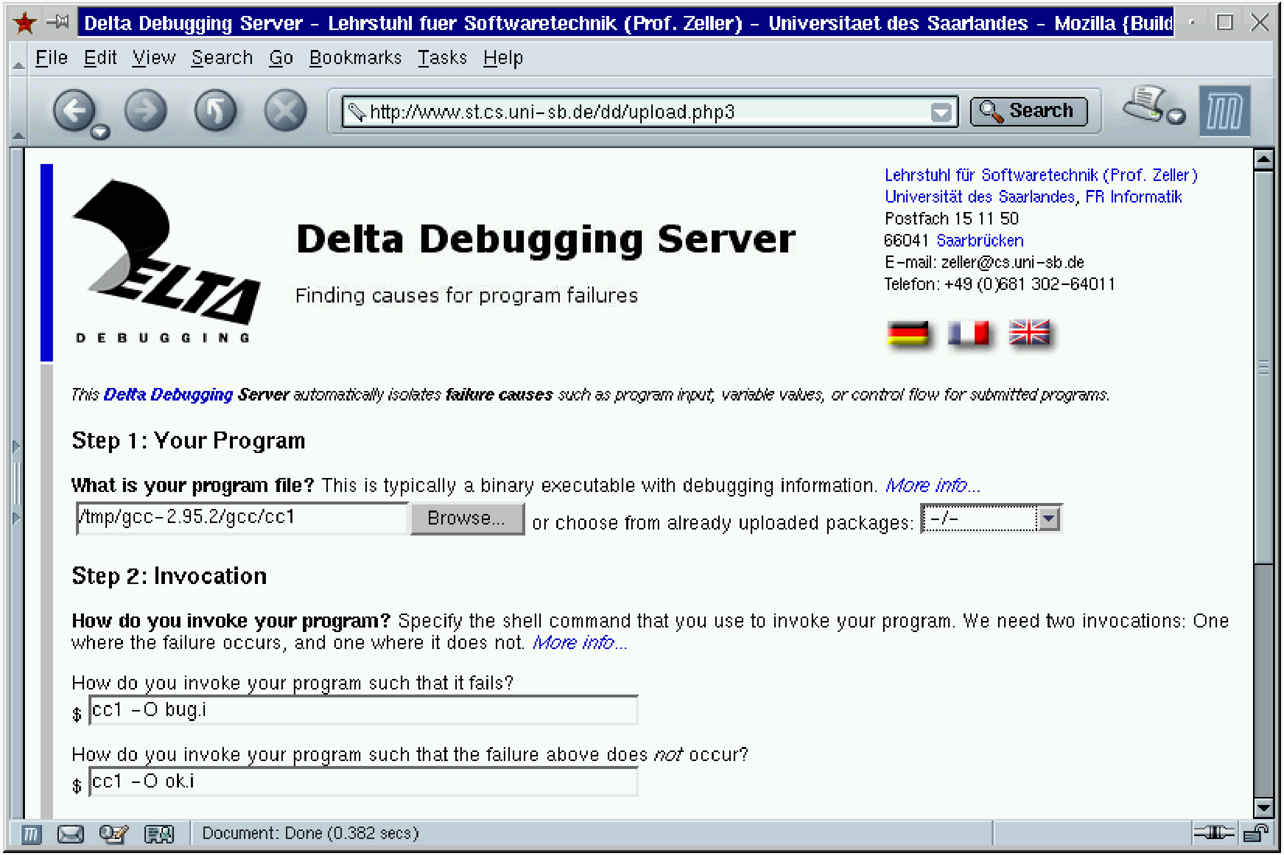
Visualization does not scale. See GCC graph with 43,000 nodes.

Program analysis scales badly. In real programs, 80–85% are possible causes of a variable value – with static and dynamic analysis!

Real code – a challenge. Real programs are *opaque, parallel, distributed, dynamic, multilingual*.

Can *experimental approaches* like Delta Debugging help?

You press the Button



Conclusion

Delta Debugging

- provides *automatic* and *precise* isolation of failure causes (= failure-inducing differences)
- *automatic* = automatic test is only requirement
- *precise* = much higher precision than program analysis

Determining Failure Causes

- requires a working run as reference
- must be able to capture and alter circumstances of the run
- does not require further knowledge about the program

`http://www.st.cs.uni-sb.de/dd/`

Read More

Automated Debugging: Are We Close?
(A. Zeller) IEEE Computer, November 2001, pp. 26–31.

Simplifying and Isolating Failure-Inducing Input.
(A. Zeller + R. Hildebrandt) IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183–200.

Isolating Failure-Inducing Thread Schedules.
(J.-D. Choi + A. Zeller) Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rome, July 2002.

Yesterday, my program worked. Today, it does not. Why?
(A. Zeller) Proc. ACM SIGSOFT Conference (ESEC/FSE), Toulouse, September 1999, Springer LNCS 1687, pp. 253–267.

Automated Debugging.
(A. Zeller) Morgan Kaufmann Publishers, Spring 2003.

http://www.st.cs.uni-sb.de/dd/

More Case Studies

Event	Edges	Vertices	Deltas	Tests
sample at *main*	26	26	12	4
sample at *shell_sort*	26	26	12	7
sample at *sample.c*:37	26	26	12	4
cc1 at *main*	27139	27159	1	0
cc1 at *combine_instructions*	42991	44290	871	44
cc1 at *if_then_else_cond*	47071	48473	1224	15
bison at *open_files*	431	432	2	2
bison at *initialize_conflicts*	1395	1445	431	42
diff at *analyze.c*:966	413	446	109	9
diff at *analyze.c*:1006	413	446	99	10
gdb at *main.c*:615	32455	33458	1	0
gdb at *exec.c*:320	34138	35340	18	7

In all cases, exactly one variable was the failure cause.

Perspectives

- ✔ Approach has been filed for patent (Bayernpatent/FhG)
- ✔ *Ernst-Denert-Preis 2001* for Diploma thesis (R. Hildebrandt)
- ✔ Automatic narrowing of *cause transitions:*
  "From when is a[0] relevant and a[2] no more?"
- ✔ Combination with *program analysis:*
  "What *could* have caused the variable value?"
- ✔ Book "Automated Debugging"
  dpunkt/Morgan Kaufmann, Spring 2003
- ✔ *Case studies* (many + large + complex)

Delta Debugging: Applications

Failure-inducing Input.

Effect: *Mozilla crashes when printing Bugzilla web page.*

Cause: A <SELECT>-Tag in the HTML input.

Failure-inducing Code Changes.

Effect: *After upgrading from GDB 4.16 to GDB 4.17, DDD ignores program arguments.*

Cause: Text "`arguments`" changed to "`arg list`".

Failure-inducing Thread Switches. (with IBM Research)

Effect: *The Java program* `raytracer` *sometimes works, sometimes it does not.*

Cause: *Data race* at 33rd thread switch.

Failure-inducing Program States.

Effect: *GCC crashes when compiling* `bug.c`*.*

Cause: Cycle in the RTL tree after optimizing $+0.0$

The Dangers of Abstraction

A true story: The code

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

prints "b = 0" – a failure.

Variable b depends on a, but is a a cause?

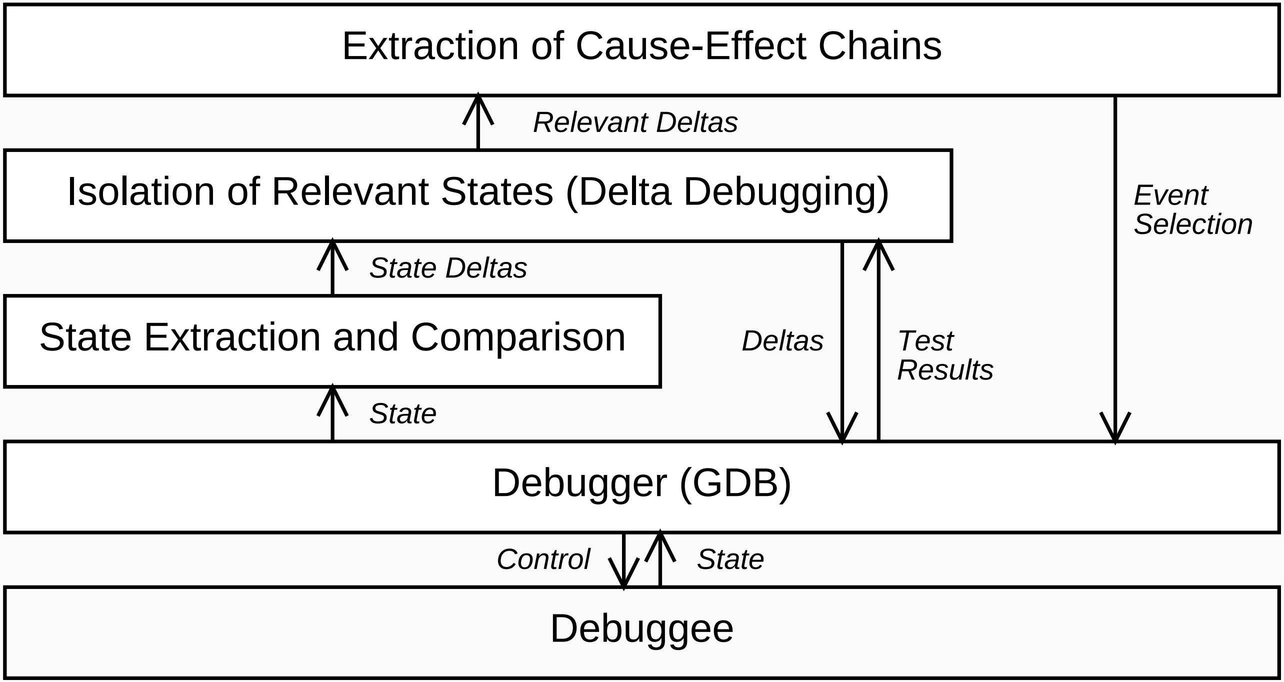
No, since the `printf` format does not match the type of b:

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

Whatever the value of a und b,
this code will *always* print "b = 0".

(Is there any *general* program analysis which detects this?)

The HOWCOME Prototype



The End

Any questions?