Why does my Program fail?

Causes and effects in computer programs

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken
Consider the following C program:

```c
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] + 1.0);
    }
    return z[n];
}
```

Compiling `bug.c`, the GNU compiler (GCC) crashes:

```
linux$ gcc-2.95.2 -O bug.c
gcc: Internal error: program cc1 got fatal signal 11
```

What’s the error that causes this failure?
Errors

What’s the error in GCC?

An error is a deviation from what’s correct, right, or true. — IEEE Standard Glossary of SE Terminology

To prove that something is an error, we must show the deviation:

• simple for the failure in question
• hard for the program code

General technique: Deduction—reasoning from the abstract (code) to the concrete (run): static analysis, verification, . . .

Where does GCC deviate from—what?
Causes

What’s the cause for the GCC failure?

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 that

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

General technique: *Experimentation*—constructing a *theory* from a series of experiments (runs)

*Can’t we automate experimentation?*
Isolating Failure Causes

*Delta Debugging* automatically isolates the *failure-inducing difference* in the GCC input:

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
</tbody>
</table>
Isolating Failure Causes

**Delta Debugging** automatically isolates the failure-inducing difference in the GCC input:

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td></td>
</tr>
</tbody>
</table>
## Isolating Failure Causes

*Delta Debugging* automatically isolates the failure-inducing difference in the GCC input:

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>double <code>bug(...)</code> { int <code>i</code>, <code>j</code>; <code>i</code> = 0; for (...) { ...} ...}</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>double <code>bug(...)</code> { int <code>i</code>, <code>j</code>; <code>i</code> = 0; for (...) { ...} ...}</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>double <code>bug(...)</code> { int <code>i</code>, <code>j</code>; <code>i</code> = 0; for (...) { ...} ...}</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Isolating Failure Causes

**Delta Debugging** automatically isolates the failure-inducing difference in the GCC input:

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✗</td>
</tr>
</tbody>
</table>
## Isolating Failure Causes

**Delta Debugging** automatically isolates the failure-inducing difference in the GCC input:

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>double <code>bug(...)</code> { int i, j; i = 0; for (...) { ... } ... }</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>double <code>bug(...)</code> { int i, j; i = 0; for (...) { ... } ... }</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>double <code>bug(...)</code> { int i, j; i = 0; for (...) { ... } ... }</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>double <code>bug(...)</code> { int i, j; i = 0; for (...) { ... } ... }</td>
<td>✔</td>
</tr>
</tbody>
</table>
Isolating Failure Causes

*Delta Debugging* automatically isolates the failure-inducing difference in the GCC input:

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>double <code>bug(...)</code> { int i, j; i = 0; for (...) { ... } ...}</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>double <code>bug(...)</code> { int i, j; i = 0; for (...) { ... } ...}</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>double <code>bug(...)</code> { int i, j; i = 0; for (...) { ... } ...}</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>double <code>bug(...)</code> { int i, j; i = 0; for (...) { ... } ...}</td>
<td>✔</td>
</tr>
<tr>
<td>5</td>
<td>double <code>bug(...)</code> { int i, j; i = 0; for (...) { ... } ...}</td>
<td>✔</td>
</tr>
</tbody>
</table>
**Isolating Failure Causes**

*Delta Debugging* automatically isolates the failure-inducing difference in the GCC input:

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✘</td>
</tr>
<tr>
<td>2</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
<tr>
<td>5</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✘</td>
</tr>
</tbody>
</table>
Isolating Failure Causes

*Delta Debugging* automatically isolates the *failure-inducing difference* in the GCC input:

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... }...}</code></td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td><code>... z[i] = z[i] * (z[0] + 1.0); ...</code></td>
<td>✗</td>
</tr>
<tr>
<td>18</td>
<td><code>... z[i] = z[i] * (z[0] + 1.0); ...</code></td>
<td>✔</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✔</td>
</tr>
</tbody>
</table>

+ 1.0 is the failure cause – after only 19 tests (≈ 2 seconds).
What’s going on in GCC?

The difference + 1.0 is just the beginning of a cause-effect chain within the GCC run.
What’s going on in GCC?

The difference $+1.0$ is just the beginning of a *cause-effect chain* within the GCC run.
What’s going on in GCC?

The difference $+ 1.0$ is just the beginning of a cause-effect chain within the GCC run.
What’s going on in GCC?

The difference $+1.0$ is just the beginning of a cause-effect chain within the GCC run.
What’s going on in GCC?

The difference \(+1.0\) is just the beginning of a *cause-effect chain* within the GCC run.

To fix the failure, we must *break* this cause-effect chain.
Tracing Data Flow

Classical *program analysis* traces how data propagates in programs.

Requires complete knowledge about entire code and its semantics ⇒ OK for small, isolated, managed programs.

But: Real programs are *opaque, parallel, distributed, dynamic, multilingual*
Tracing Data Flow

Classical *program analysis* traces how data propagates in programs.

Requires complete knowledge about entire code and its semantics ⇒ OK for small, isolated, managed programs.

But: Real programs are *opaque, parallel, distributed, dynamic, multilingual*—or simply obscure:

```c
struct foo {
    int tp, len;
    union {
        char c[1];
        int i[1];
        double d[1];
    }
}
```
Tracing Data Flow

Classical *program analysis* traces how data propagates in programs.

Requires complete knowledge about entire code and its semantics ⇒ OK for small, isolated, managed programs.

But: Real programs are *opaque, parallel, distributed, dynamic, multilingual*—or simply obscure:

```c
struct foo {
    int tp, len;
    union {
        char c[1];
        int i[1];
        double d[1];
    }
};
```

// Allocate string
int len = 200;
int bytes = len + 2 * sizeof(int);
foo *x = (foo *)malloc(bytes);
x->tp = STRING;
x->len = len;
strncpy(x->c, "Some string", len);
```
Small Cause, Big Effect

Another problem—differences accumulate during execution:

Input
.
.
.
.
.
.
.
.
.
Program State (= Variables)
.
.
.
Final State
✘+1.0

How do we isolate the relevant state differences?
**Relevant State Differences**

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*

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
</tbody>
</table>
Relevant State Differences

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

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✓</td>
</tr>
</tbody>
</table>
Relevant State Differences

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`

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✘</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td></td>
</tr>
</tbody>
</table>
**Relevant State Differences**

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*

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>curInsn_uid</th>
<th>last_linenumber</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
</tbody>
</table>
## Relevant State Differences

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`

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
</tbody>
</table>
**Relevant State Differences**

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`

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4e4</td>
<td>?</td>
</tr>
</tbody>
</table>
**Relevant State Differences**

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*

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4a0</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4e4</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✓</td>
</tr>
</tbody>
</table>
**Relevant State Differences**

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*

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_lineno</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4e4</td>
<td>❓</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
</tbody>
</table>

Consequence: determine and apply *structural differences!*
The GCC Memory Graph

Our IGOR prototype extracts the program state as graph: Vertices are variables, edges are references
Structural Differences

IGOR can compute structural graph differences:

\( \Delta_{15} \) creates a variable, \( \Delta_{20} \) deletes another
The Process in a Nutshell

Failing Run

Passing Run
The Process in a Nutshell

Failing Run

Passing Run
The Process in a Nutshell

Failing Run

Passing Run
**Relevant State Differences**

IGOR examines the state of cc1 in *combine_instructions*: 871 nodes (= variables) are different
Relevant State Differences

IGOR examines the state of cc1 in `combine_instructions`: 871 nodes (= variables) are different.

Only one variable causes the failure:

```c
$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, IGOR has determined these failure causes:

1. Execution reaches main.
   Since the program was invoked as “cc1 -O fail.i”,
   variable argv[2] is now “fail.i”.
The GCC Cause-Effect Chain

After 59 tests, IGOR has determined these failure causes:

1. Execution reaches main.
   Since the program was invoked as “cc1 -O fail.i”, variable argv[2] is now “fail.i”.

2. Execution reaches combine_instructions.
The GCC Cause-Effect Chain

After 59 tests, IGOR has determined these failure causes:

1. Execution reaches main.
   Since the program was invoked as “cc1 -O fail.i”,
   variable argv[2] is now “fail.i”.

2. Execution reaches combine_instructions.
   Since argv[2] was “fail.i”,
   variable *first_loop_store_insn→fld[1].rtx→fld[1].rtx→
   fld[3].rtx→fld[1].rtx is now ⟨new rtx_def⟩.

3. Execution reaches if_then_else_cond (95th hit).
   Since *first_loop_store_insn→fld[1].rtx→fld[1].rtx→
   fld[3].rtx→fld[1].rtx was ⟨new rtx_def⟩,
   variable link→fld[0].rtx→fld[0].rtx is now link.
The GCC Cause-Effect Chain

After 59 tests, IGOR has determined these failure causes:

1. Execution reaches `main`.  
   Since the program was invoked as “cc1 -O fail.i”, variable `argv[2]` is now “`fail.i`”.

2. Execution reaches `combine_instructions`.  

3. Execution reaches `if_then_else_cond (95th hit)`.  
   Since `*first_loop_store_insn->fld[1].rtx->fld[1].rtx->fld[3].rtx->fld[1].rtx` was `(new rtx_def)`, variable `link->fld[0].rtx->fld[0].rtx` is now `link`.

4. Execution ends.  
   Since variable `link->fld[0].rtx->fld[0].rtx` was link, the program now terminates with a `SIGSEGV` signal. The program fails.

Total running time: 6 seconds
The GCC Cause-Effect Chain

After 59 tests, IGOR has determined these failure causes:

1. **Execution reaches `main`.**
   Since the program was invoked as “cc1 -O fail.i”, variable `argv[2]` is now “`fail.i`”.

2. **Execution reaches `combine_instructions`.**

3. **Execution reaches `if_then_else_cond (95th hit)`.**
   Since `*first_loop_store_insn→fld[1].rtx→fld[1].rtx→fld[3].rtx→fld[1].rtx` was `<new rtx_def>`, variable `link→fld[0].rtx→fld[0].rtx` is now `link`.

4. **Execution ends.**
   Since variable `link→fld[0].rtx→fld[0].rtx` was link, the program now **terminates with a SIGSEGV signal.**
   The program fails.

Total running time: 6 seconds (+ 90 minutes of GDB overhead)
Causes vs. Errors

Every failure is caused by some error. But where is the error?

Deduction finds errors—but to prove that some error causes a given failure requires a fix.

Where’s the technology that fixes errors?
Causes vs. Errors

Every failure is caused by some error. But where is the error?

**Deduction** finds errors—but to prove that some error causes a given failure requires a fix.

*Where’s the technology that fixes errors?*

**Experimentation** finds causes—but to prove that some failure cause is an error requires a *full specification*.

*Without specification, there are no errors—only surprises.*
Causes vs. Errors

Every failure is caused by some error. But where is the error?

**Deduction** finds errors—but to prove that some error causes a given failure requires a fix.

*Where’s the technology that fixes errors?*

**Experimentation** finds causes—but to prove that some failure cause is an error requires a *full specification*.

*Without specification, there are no errors—only surprises.*

You don’t know you found the error until it’s fixed:

- Absence of failure proves that the error caused the failure
- The fixed version is (hopefully) correct, right, and true
Isolating the Error

We can narrow down the error by (manually) distinguishing erroneous and non-erroneous causes.
Isolating the Error

We can narrow down the error by (manually) distinguishing erroneous and non-erroneous causes.
**Isolating the Error**

We can narrow down the error by (manually) distinguishing erroneous and non-erroneous causes.
**Isolating the Error**

We can narrow down the error by (manually) distinguishing erroneous and non-erroneous causes.
Isolating the Error

We can narrow down the **error** by (manually) distinguishing **erroneous** and **non-erroneous** causes.
Isolating the Error

We can narrow down the error by (manually) distinguishing erroneous and non-erroneous causes.

Bad alias in distributive law in lines 4013–4019; fixed in 2.95.3

\[(+ (* a b) c) \Rightarrow (* (+ a c_1) (+ b c_2)) \text{ with } c = c_1 = c_2\]
**Challenges**

How do we capture C program state accurately?

*Does p point to something, and if so, to how many of them?*

Today: Query memory allocation routines + heuristics

Future: Use program analysis, greater program state
Challenges

How do we capture C program state accurately?

*Does p point to something, and if so, to how many of them?*

Today: Query memory allocation routines + heuristics
Future: Use program analysis, greater program state

How do we determine relevant events?

*Why focus on, say, combine_instructions?*

Today: Start with backtrace of failing run
Future: Focus on anomalies + transitions; user interaction
Challenges

How do we capture C program state accurately?

*Does p point to something, and if so, to how many of them?*

Today: Query memory allocation routines + heuristics
Future: Use program analysis, greater program state

How do we determine relevant events?

*Why focus on, say, combine_instructions?*

Today: Start with backtrace of failing run
Future: Focus on anomalies + transitions; user interaction

How do we know a failure is the failure?

*Can’t our changes just induce new failures?*

Today: Outcome is “original” only if backtraces match
Future: Also match output, time, code coverage
Challenges

How do we capture C program state accurately?

*Does p point to something, and if so, to how many of them?*

Today: Query memory allocation routines + heuristics
Future: Use program analysis, greater program state

How do we determine relevant events?

*Why focus on, say, combine_instructions?*

Today: Start with backtrace of failing run
Future: Focus on anomalies + transitions; user interaction

How do we know a failure is the failure?

*Can’t our changes just induce new failures?*

Today: Outcome is “original” only if backtraces match
Future: Also match output, time, code coverage

And finally: *When does this actually work?*
Submit buggy program  

Specify invocations  

Click on “Debug it”  

Diagnosis comes via e-mail  

Up and running since Summer 2003  

56% “pinpoints the bug”  
22% “helpful insights”
Delta Debugging Plug-Ins
Delta Debugging in one Run

In a reactive program, one single run may suffice:
**Delta Debugging in one Run**

In a reactive program, one single run may suffice:

Comparing program state *at different moments in time* again reveals differences...
**Delta Debugging in one Run**

In a reactive program, one single run may suffice:

Comparing program state *at different moments in time* again reveals differences, which may be narrowed down to causes.

Applications: interactive programs, servers, device drivers...
Self-Repairing Programs

A fatal error has occurred.
Auto-repair in progress; please stand by...

13%
Self-Repairing Programs

A fatal error has occurred.
Auto-repair in progress; please stand by...

Auto-repair successful.
Some services have been turned off. (Details)
Self-Repairing Programs

A fatal error has occurred.
Auto-repair in progress; please stand by...

Auto-repair successful.
Some services have been turned off. (Details)

Sub-Pixel anti-aliasing has been turned off, as it caused a fatal error. (Reactivate)

OK
Past and Future

Past 20 years: *deduction* and *observation* techniques

Observation: 1 run

Deduction: 0 runs
Past and Future

Past 20 years: *deduction* and *observation* techniques

Next 20 years: *induction* and *experimentation*?
Conclusion

▷ We may be able to guarantee the absence of errors—but never the *absence of surprises*.

▷ Failure causes can be isolated *automatically*... 
  - if we have an automated test
  - where at least one test case passes

▷ Systematic *experimentation* can significantly *augment* “classical” program analysis.

▷ Via automation, debugging becomes a *well-understood and systematic discipline*.

▷ Book “Why does my program fail?” (MK) in Fall 2004

http://www.askigor.org/
Read More


http://www.askigor.org/
About this Presentation

This presentation was created by Andreas Zeller, Professor of Computer Science at Saarland University, Saarbrücken, Germany. Contact him at

http://www.st.cs.uni-sb.de/~zeller/

This presentation, its source code, and additional material can be downloaded at

http://www.st.cs.uni-sb.de/papers/fse2002/

This presentation is licensed under the Creative Commons Attribution License. To view a copy of this license, visit

http://creativecommons.org/licenses/by/1.0/

or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.