Microsoft Research, Redmond, Washington, 2003-10-13

Why does my Program fail?

Causes and effects in computer programs

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

A True Story

Consider the following C program:

```
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:

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

Isolating Failure Causes

Delta Debugging automatically isolates the *failure-inducing difference* in the 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

Delta Debugging automatically isolates the *failure-inducing difference* in the 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

Delta Debugging automatically isolates the *failure-inducing difference* in the 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

Delta Debugging automatically isolates the *failure-inducing difference* in the 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

Delta Debugging automatically isolates the *failure-inducing difference* in the 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

Delta Debugging automatically isolates the *failure-inducing difference* in the 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

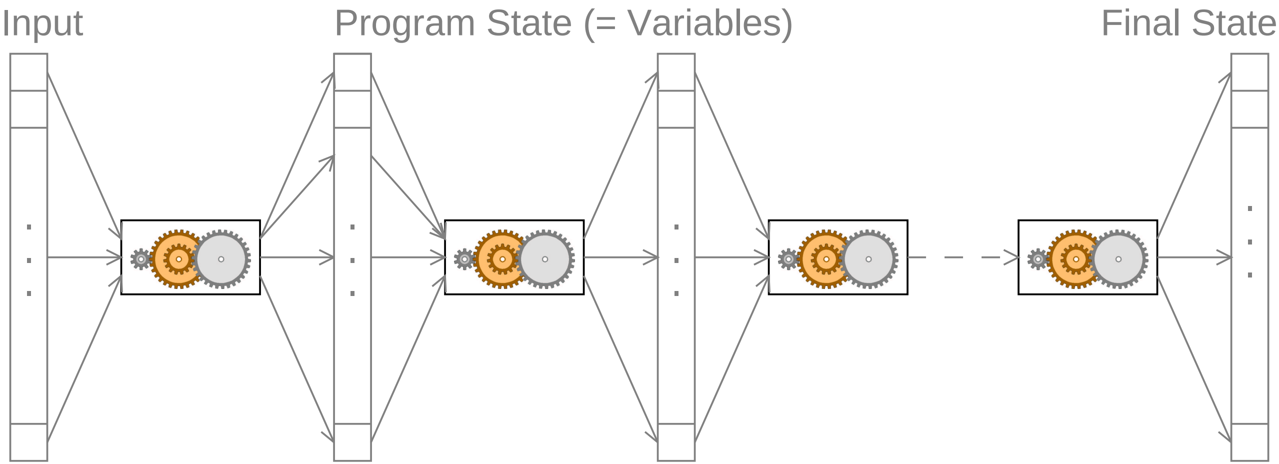
Delta Debugging automatically isolates the *failure-inducing difference* in the GCC input:

#	GCC input	test
1	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✘
5	double **bug**(...) { int i, j; $i = 0$; for (...) { ... }... }	✘
⋮		
19	... $z[i] = z[i] * (z[0] + 1.0)$; ...	✘
18	... $z[i] = z[i] * (z[0] + 1.0)$; ...	✔
⋮		
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 (...) { ... } ... }	✔

$+ 1.0$ is the failure cause – after only 19 tests ($\approx$ 2 seconds).

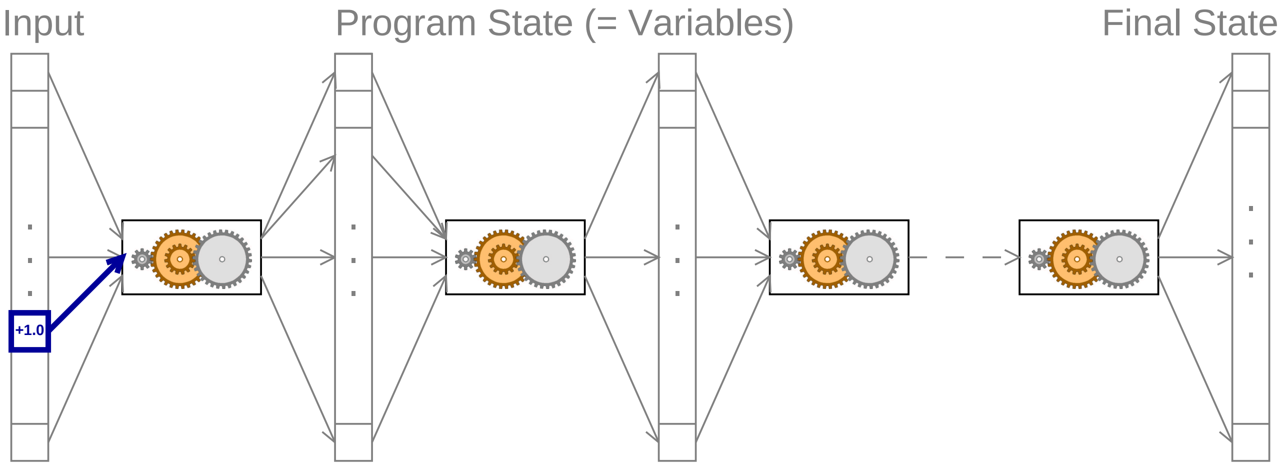
What's going on in GCC?

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



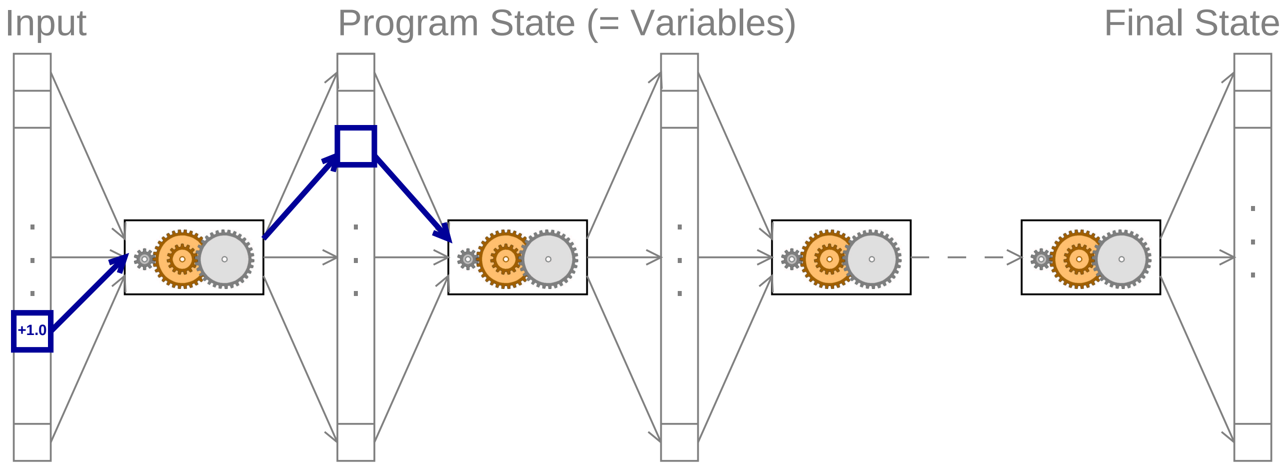
What's going on in GCC?

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



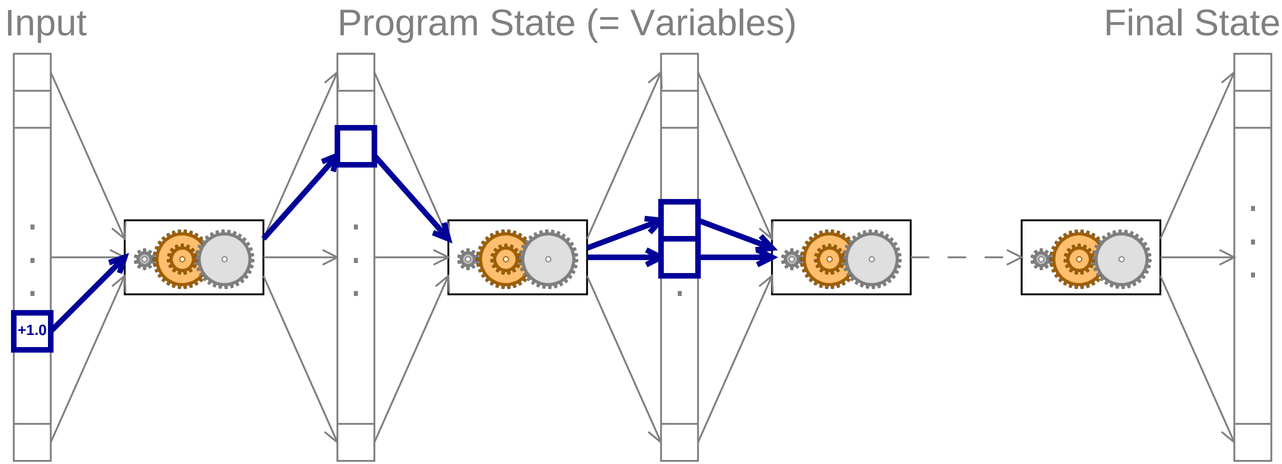
What's going on in GCC?

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



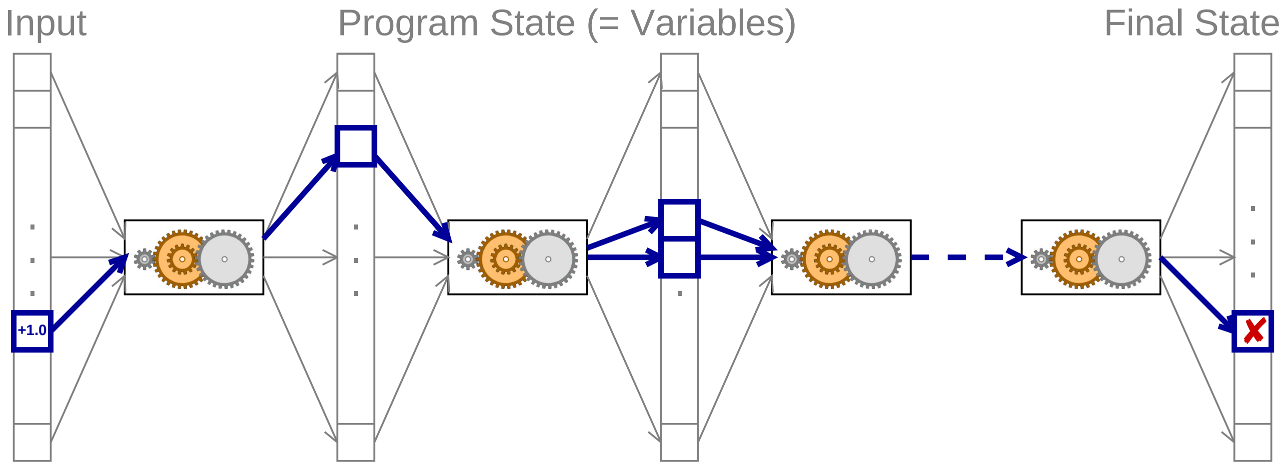
What's going on in GCC?

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:

```
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.

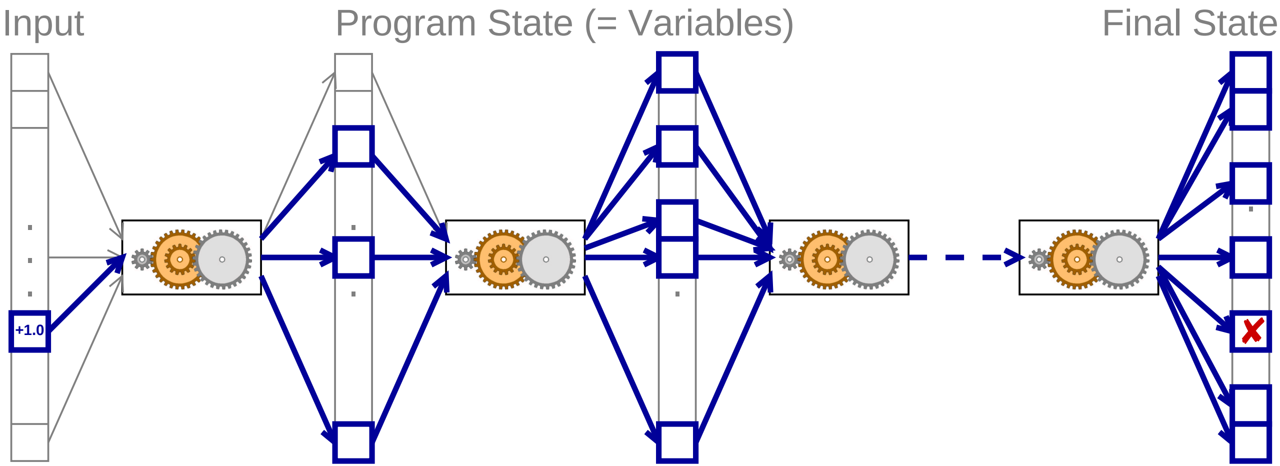
But: Real programs are *opaque, parallel, distributed, dynamic, multilingual*—or simply obscure:

```
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:



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*

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

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*

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

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*

#	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	✔

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*

#	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	✔

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*

#	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	✔

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*

#	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	✔

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*

#	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	✔

Relevant State Differences

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

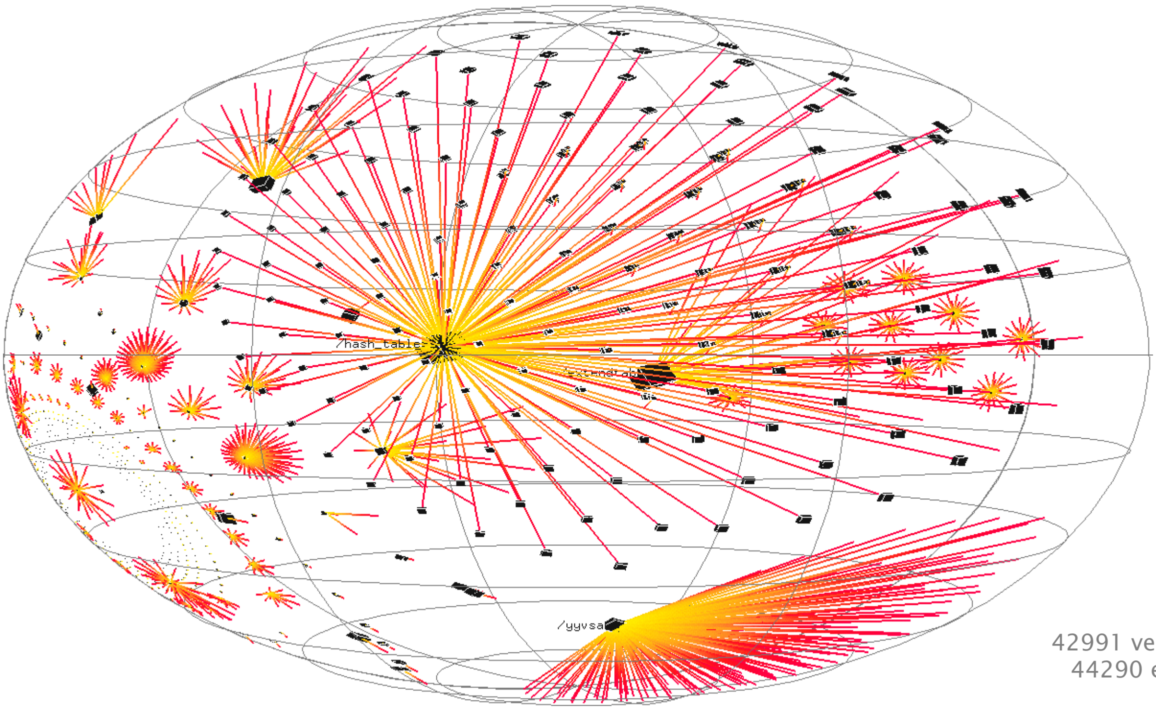
Example: GCC state in the function *combine_instructions*

#	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	✔

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*

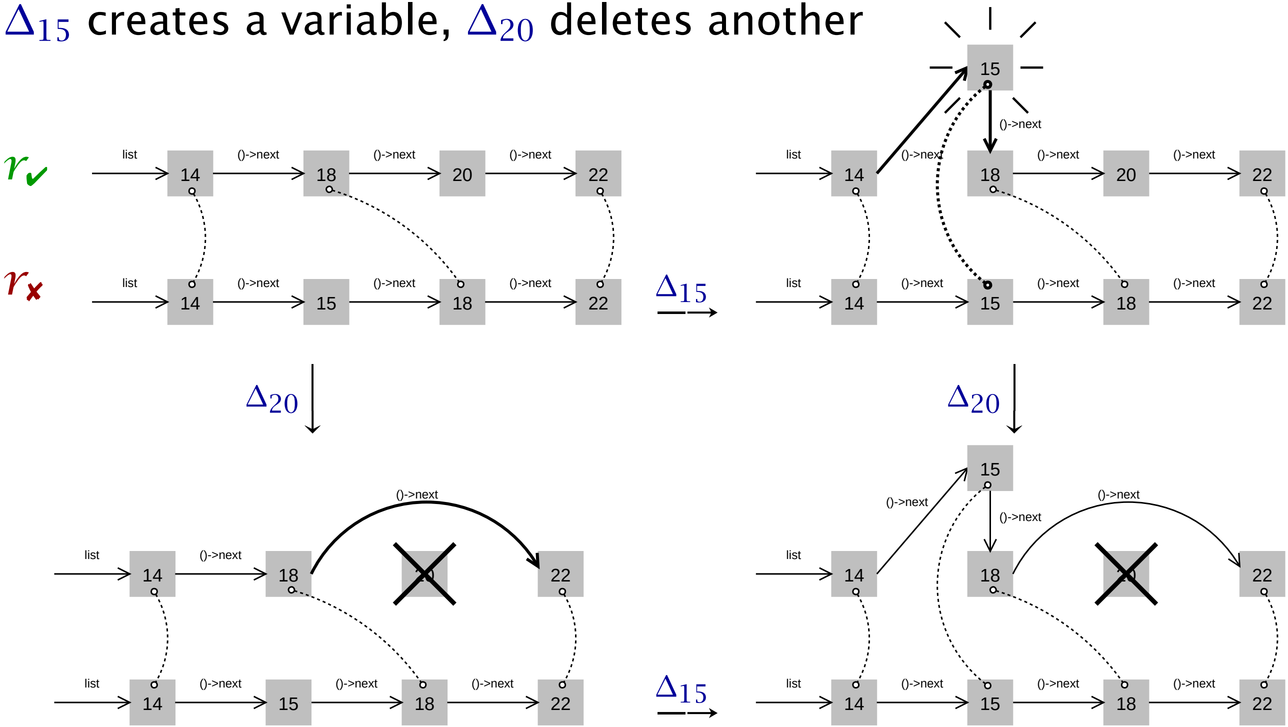


42991 vertices
44290 edges

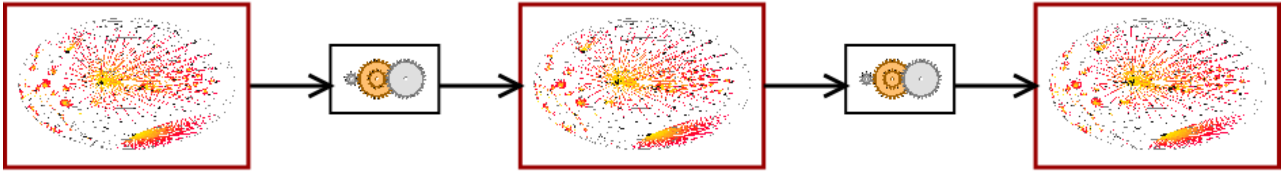
Structural Differences

IGOR can compute structural graph differences:

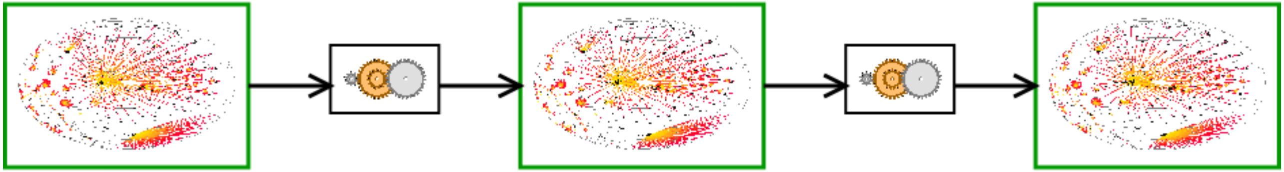
Δ_{15} creates a variable, Δ_{20} deletes another



The Process in a Nutshell

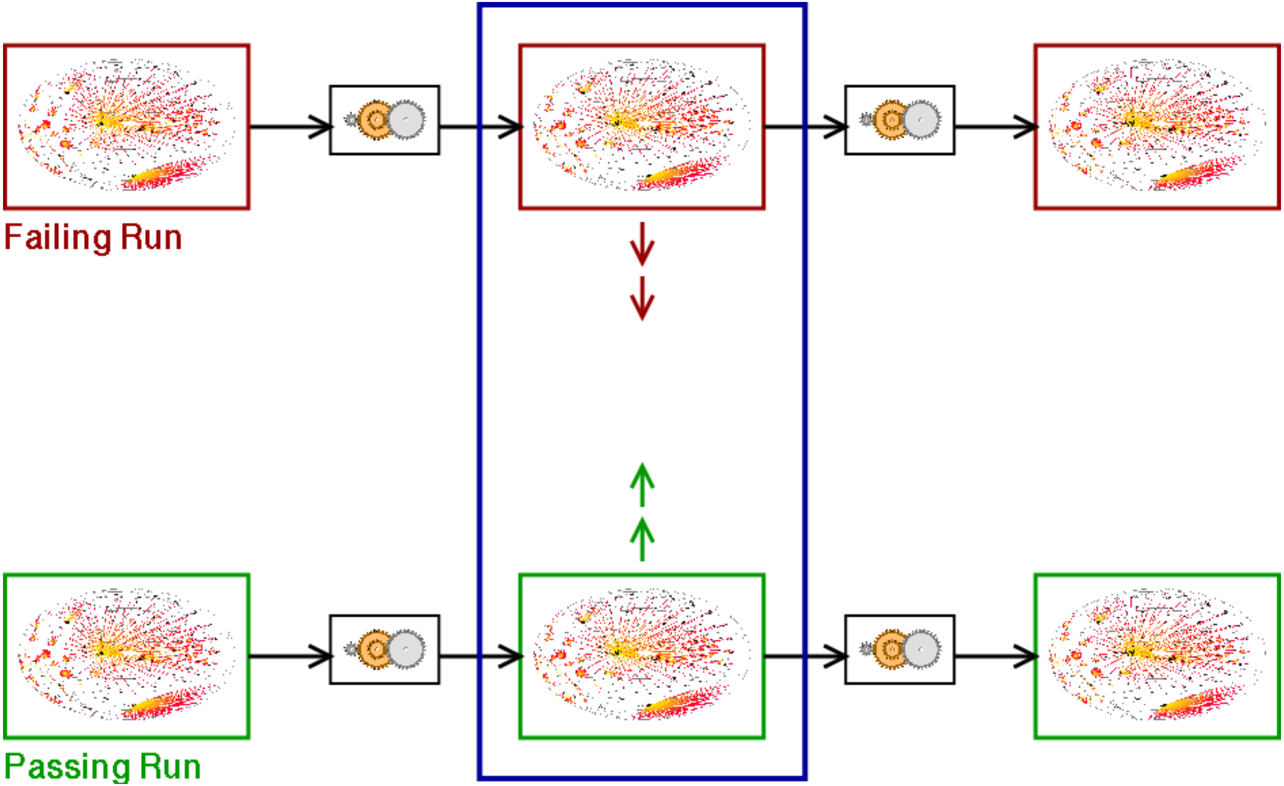


Failing Run

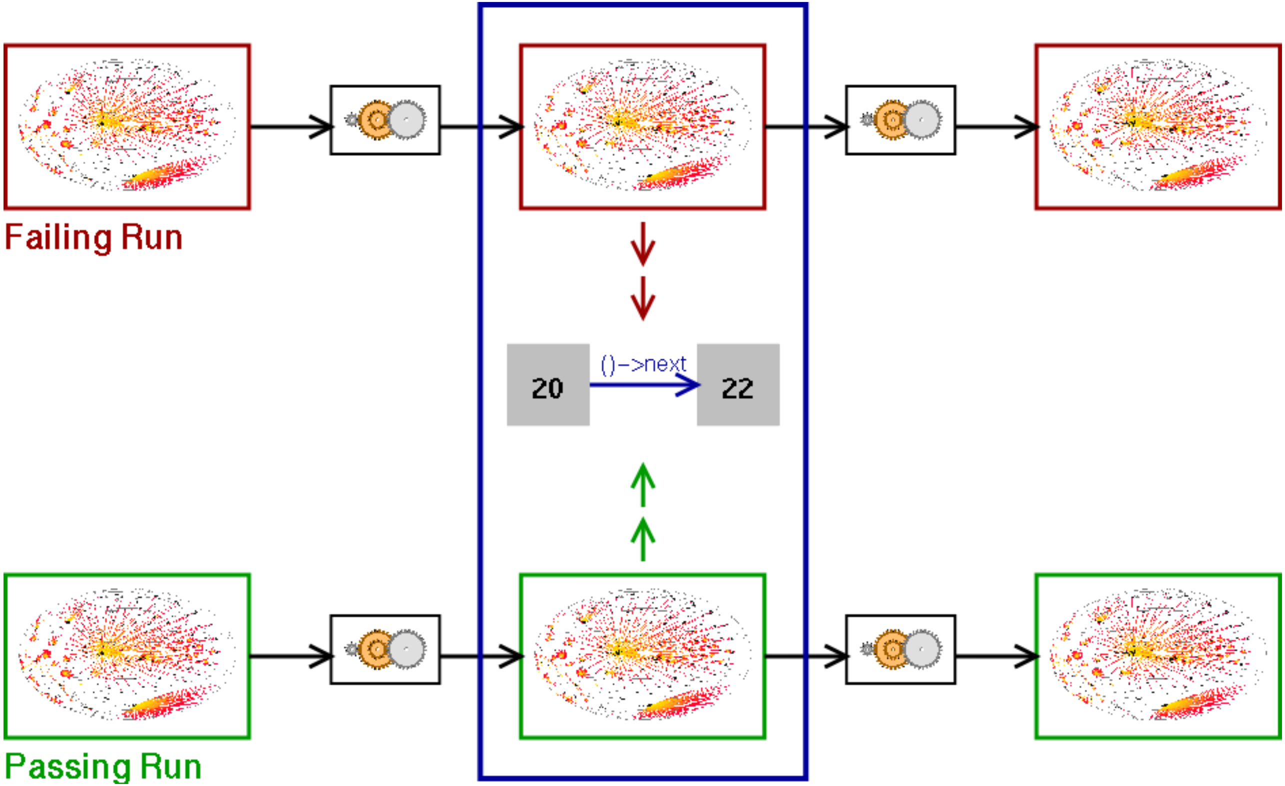


Passing Run

The Process in a Nutshell

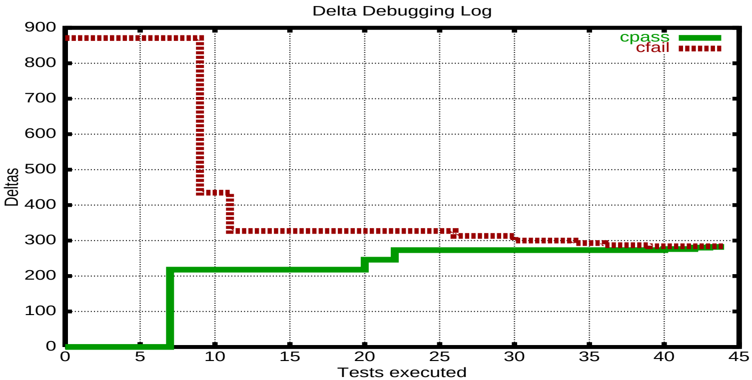


The Process in a Nutshell



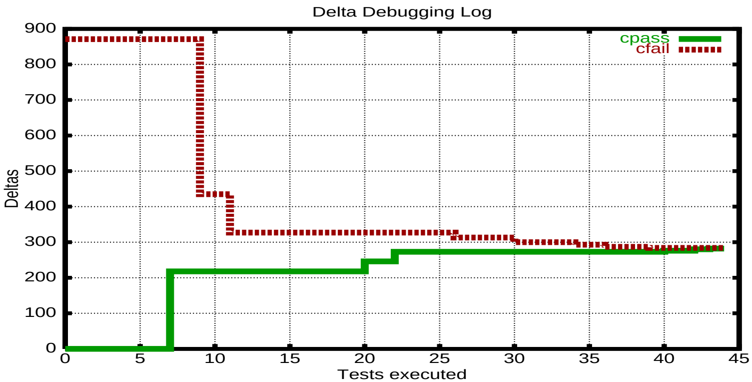
Relevant State Differences

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:

```
$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.**
   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**⟩.

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.**
   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.**

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.**
   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.**

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.*

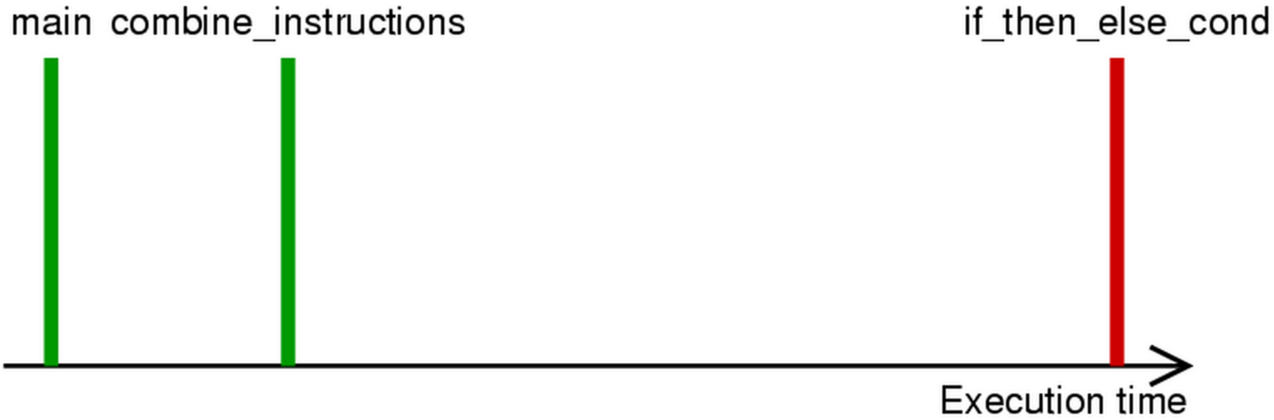
Without specification, there are no errors—only surprises.

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

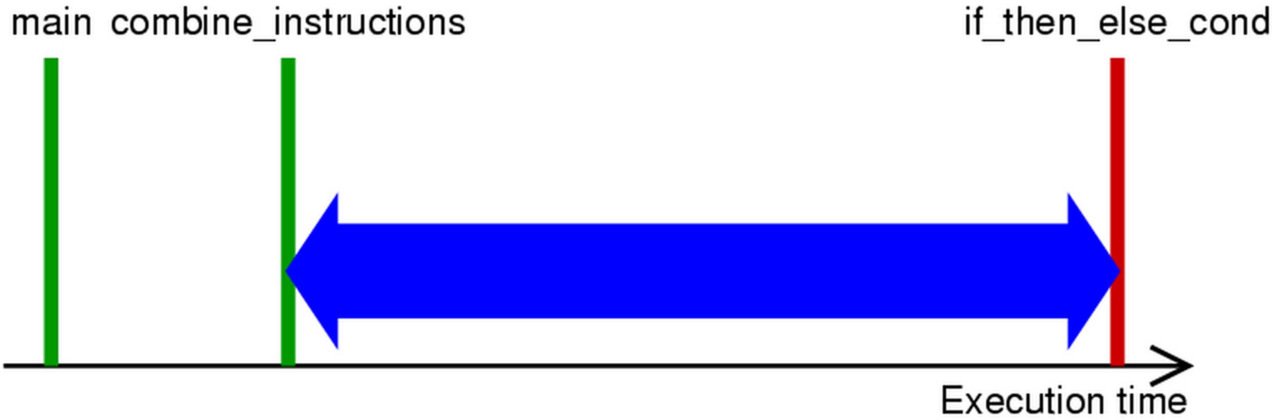
Isolating the Error

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.

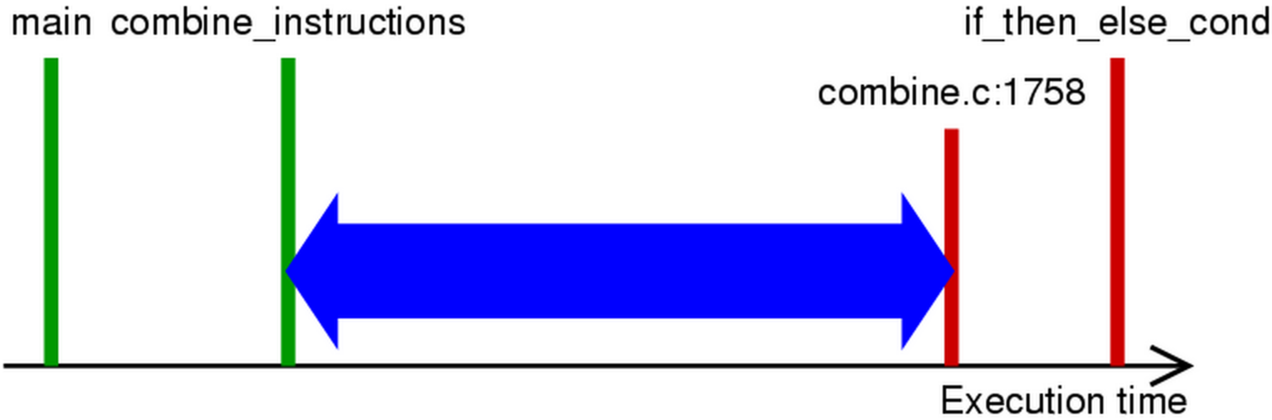


Isolating the Error

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

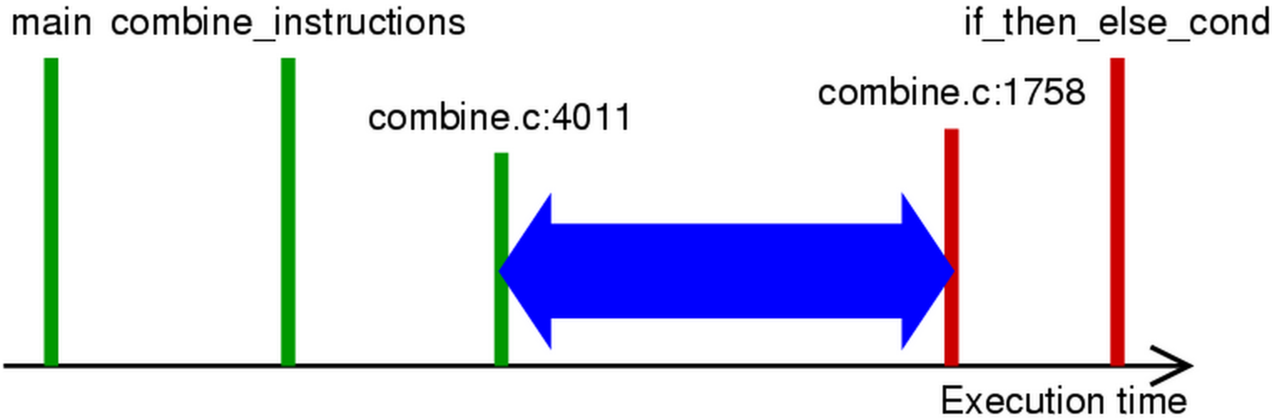
Isolating the Error

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



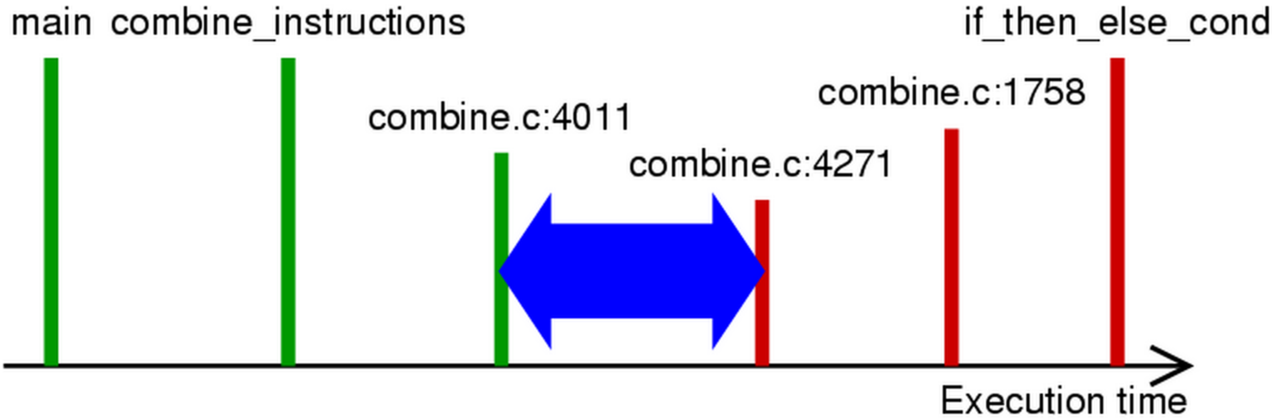
Isolating the Error

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

$(+\ (*\ a\ b)\ c) \Rightarrow (*\ (+\ a\ c_1)(+\ b\ c_2))$ 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?

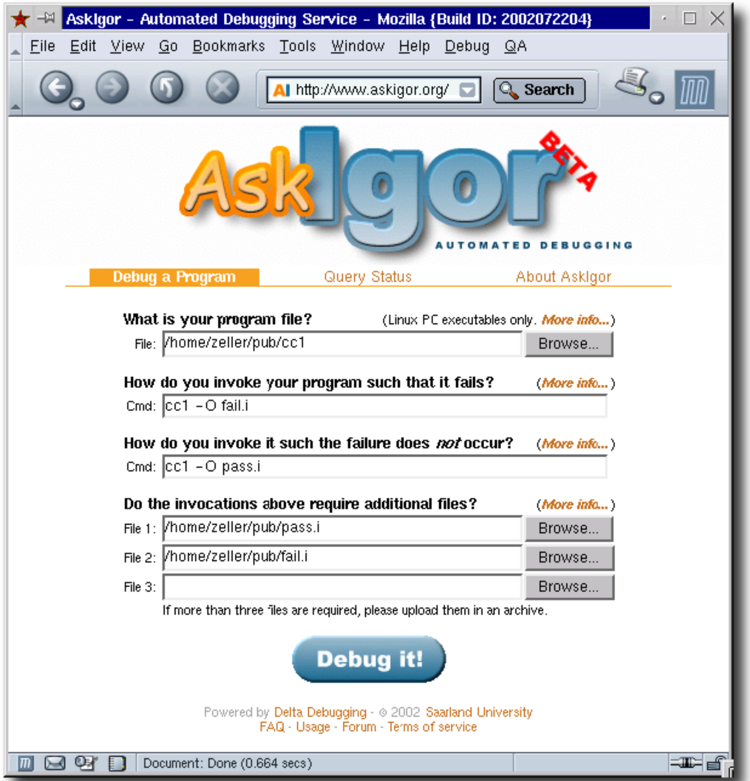
Can't our changes just induce new failures?

Today: Outcome is "original" only if backtraces match

Future: Also match output, time, code coverage

And finally: *When does this actually work?*

www.askigor.org



Submit buggy program

⇓

Specify invocations

⇓

Click on "Debug it"

⇓

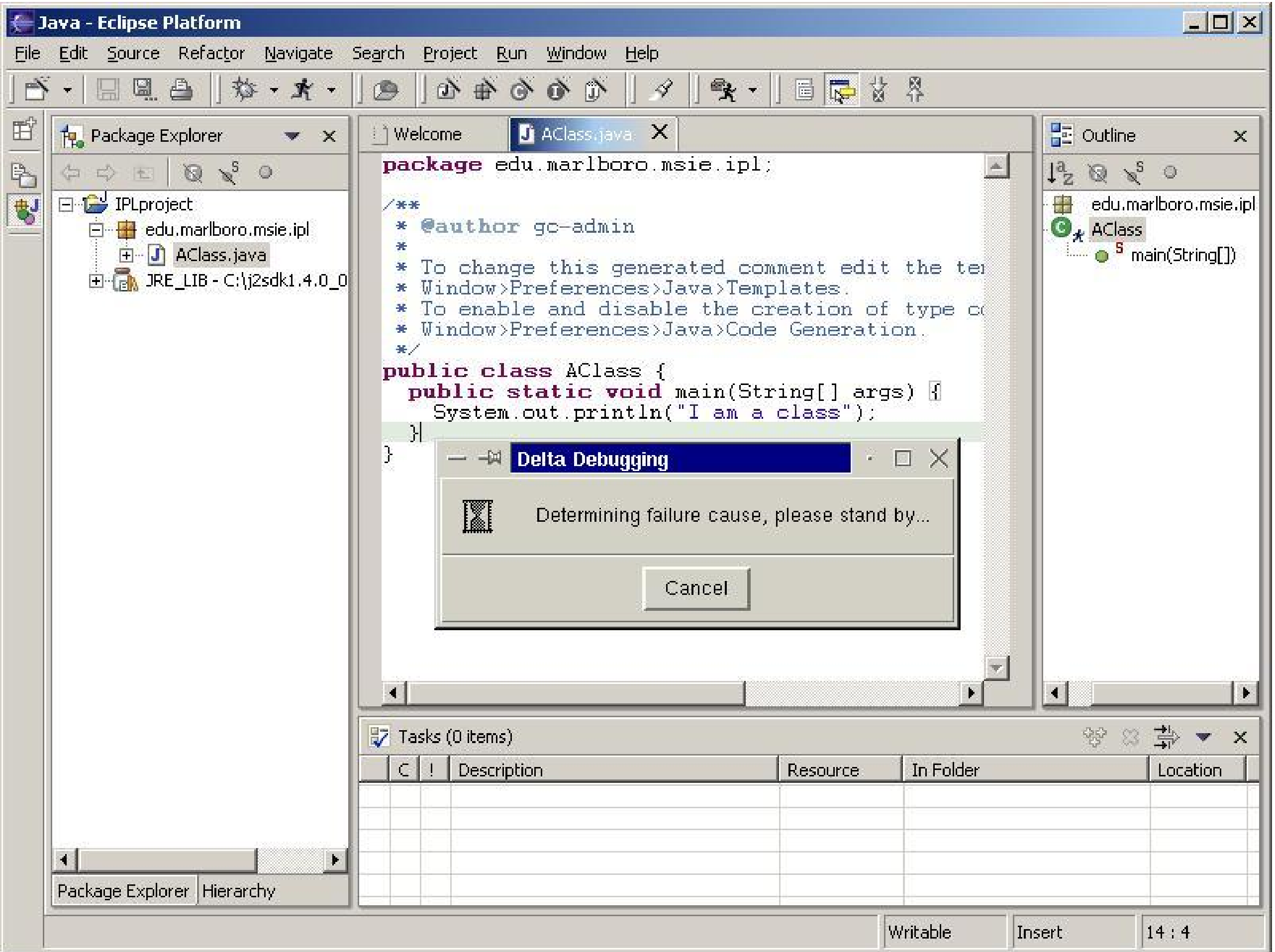
Diagnosis comes via e-mail

Up and running since Summer 2003

·

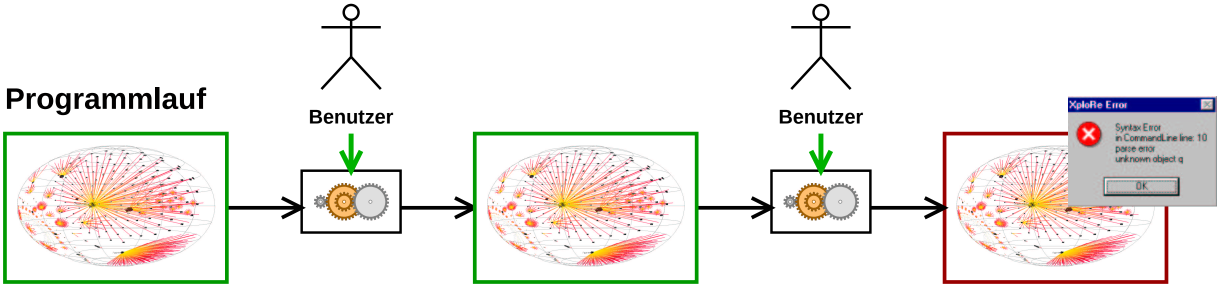
56% "pinpoints the bug"
22% "helpful insights"

Delta Debugging Plug-Ins



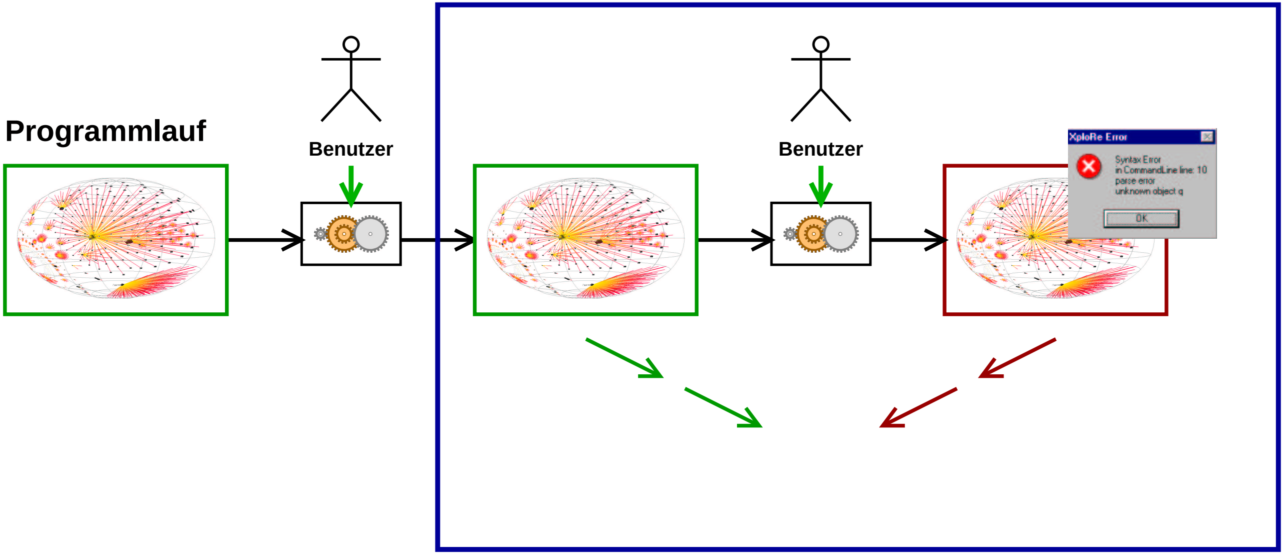
Delta Debugging in one Run

In a reactive program, one single run may suffice:



Delta Debugging in one Run

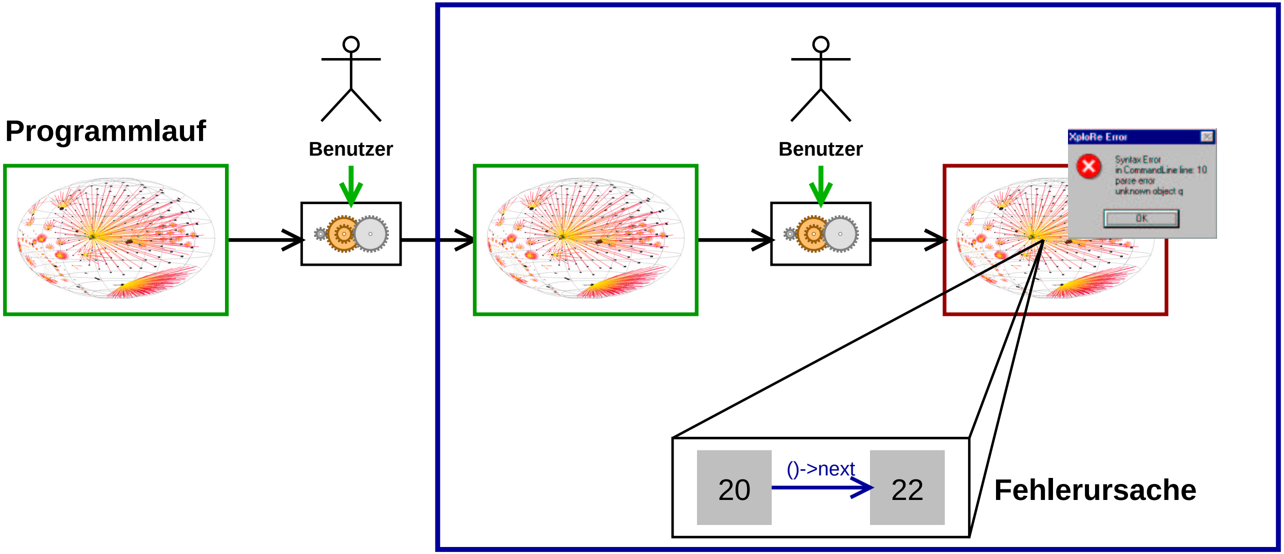
In a reactive program, one single run may suffice:



Comparing program state *at different moments in time* again reveals differences...

Delta Debugging in one Run

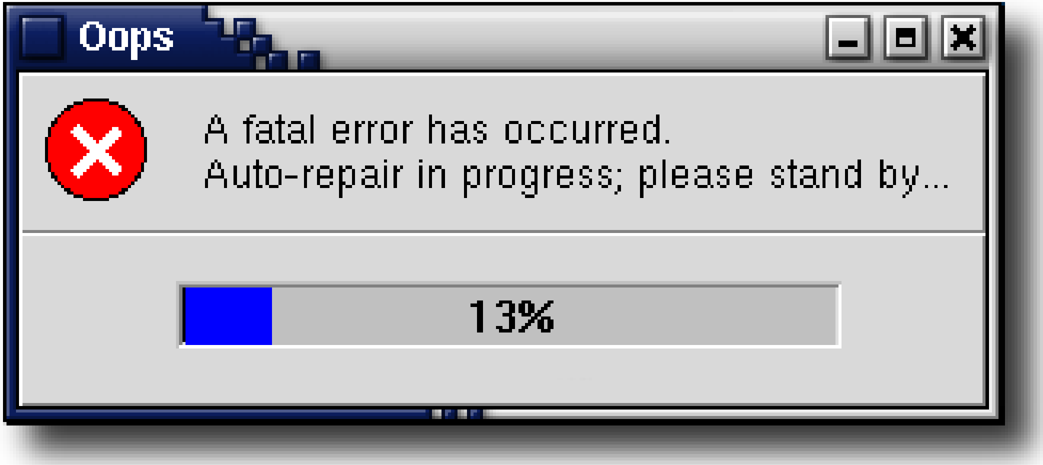
In a reactive program, one single run may suffice:



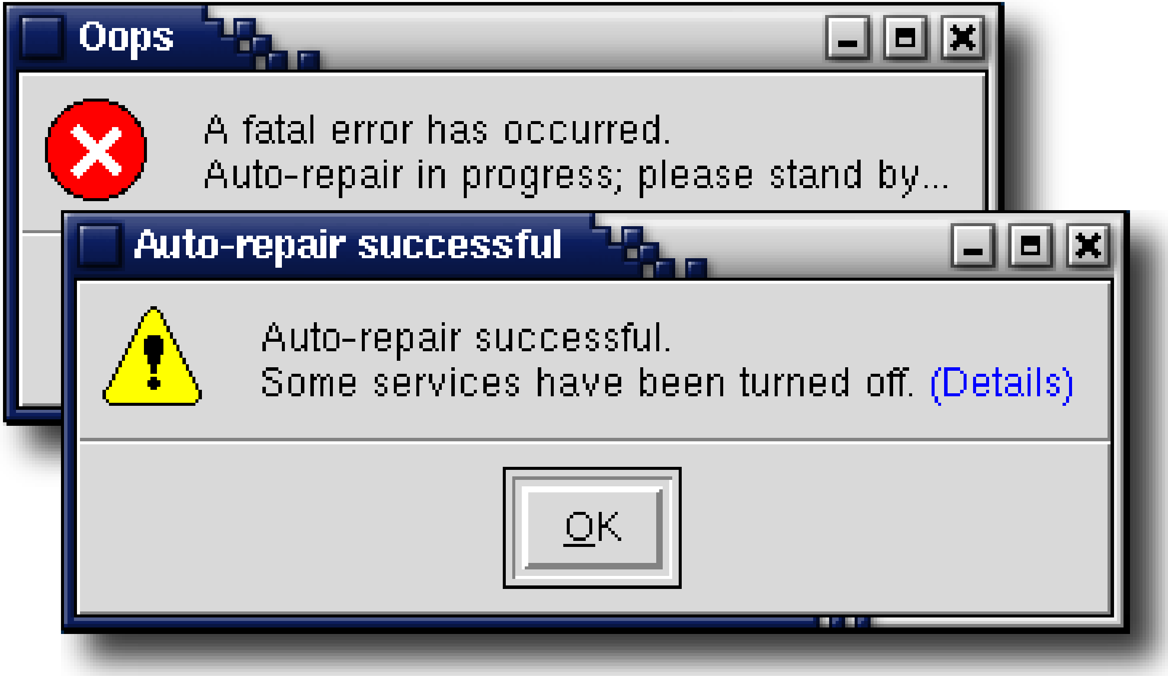
Comparing program state *at different moments in time* again reveals differences, which may be narrowed down to causes.

Applications: interactive programs, servers, device drivers. . .

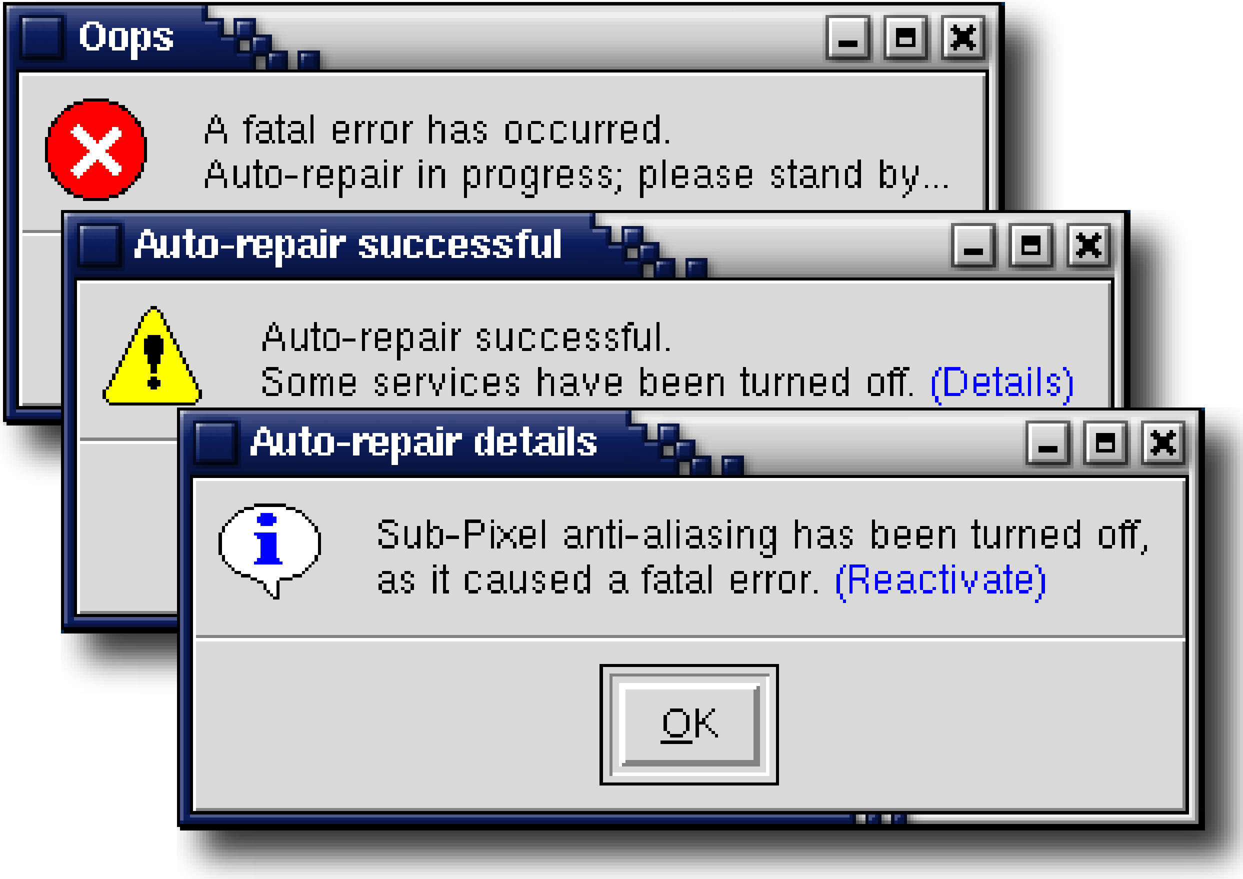
Self-Repairing Programs



Self-Repairing Programs

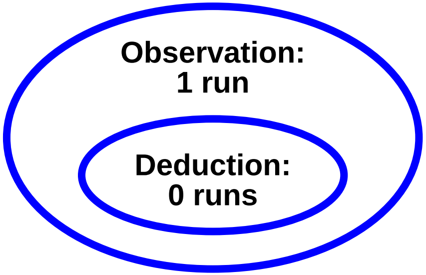


Self-Repairing Programs



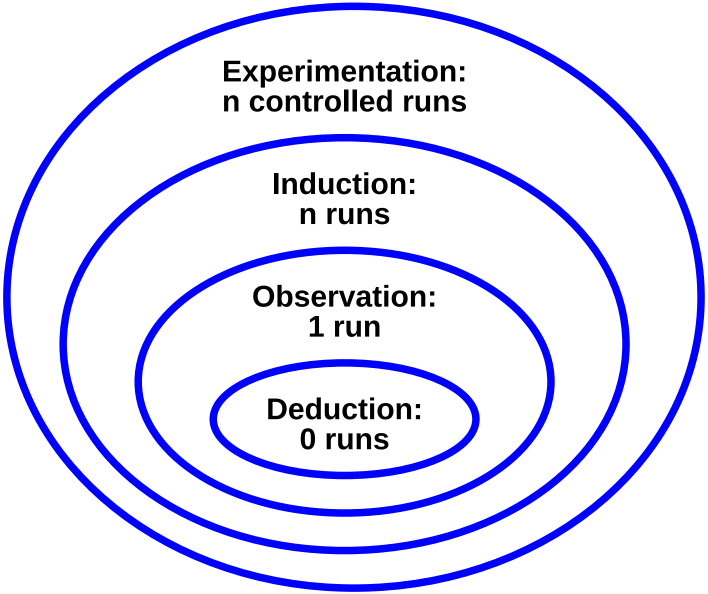
Past and Future

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



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

Why does my Program Fail? A Guide to Automated Debugging. Morgan Kaufmann Publishers, Fall 2004.

Isolating Cause-Effect Chains from Computer Programs. Proc. ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2002), Charleston, Nov. 2002.

Isolating Failure-Inducing Thread Schedules. (w/ J.-D. Choi) Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rom, July 2002.

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

Automated Debugging: Are We Close? IEEE Computer, Nov. 2001, pp. 26–31.

Visualizing Memory Graphs. (w/ T. Zimmermann) Proc. of the Dagstuhl Seminar 01211 "'Software Visualization"', May 2001. LNCS 2269, pp. 191–204.

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

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/