Lecture 19

Delta Debugging
Cooperative Bug Isolation

Today's Agenda

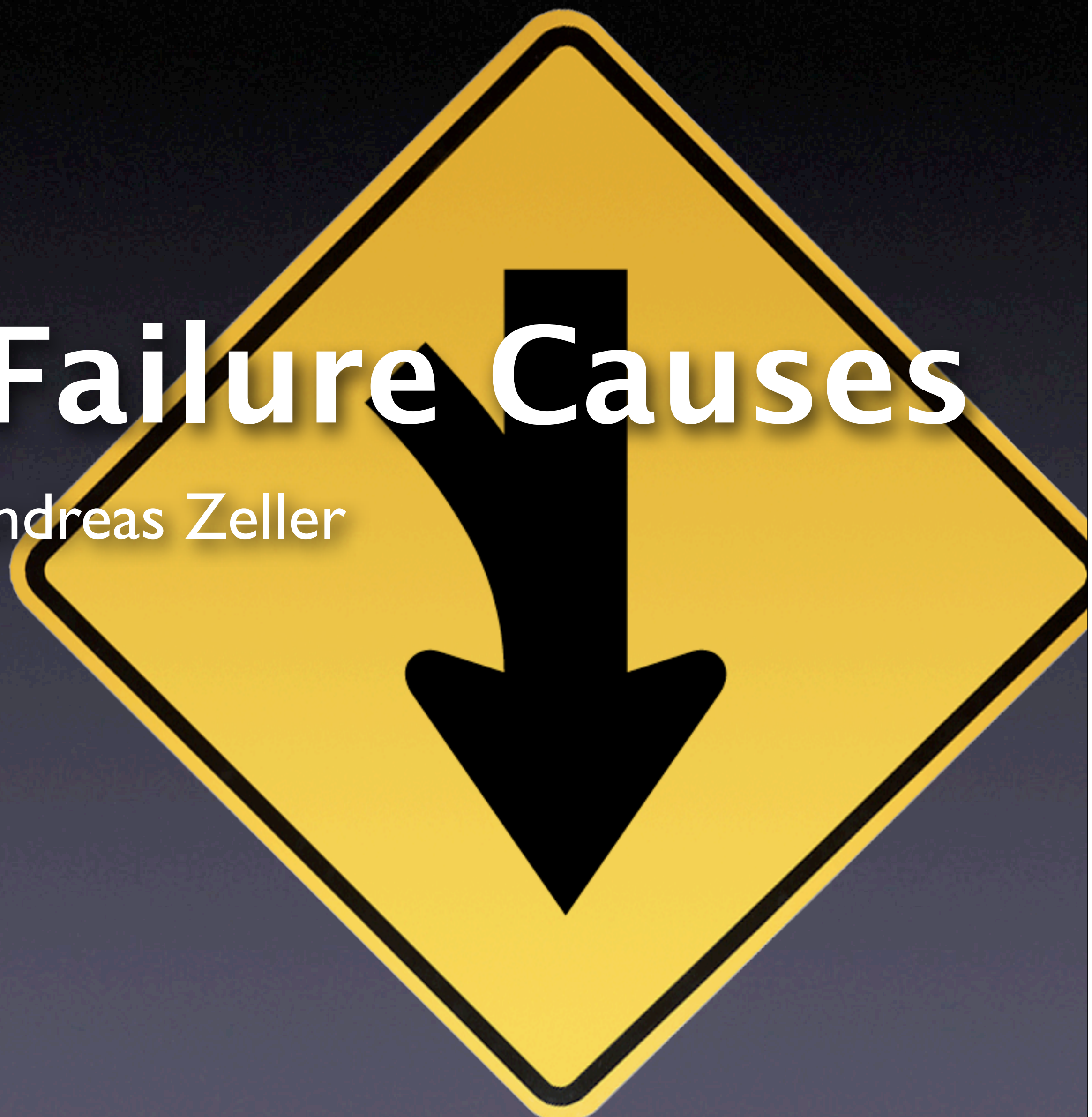
- Presentation:
  - Chris on Cooperative Bug Isolation
- Quiz on Delta Debugging

Today's Agenda

- Delta Debugging:
  - Simplifying Failure Causes => Isolating Failure Causes
  - Applications of Delta Debugging Algorithm
  - Isolating Cause and Effect Chain

Quiz: Delta Debugging

Isolating Failure Causes



Andreas Zeller

Simplifying Input

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

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

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

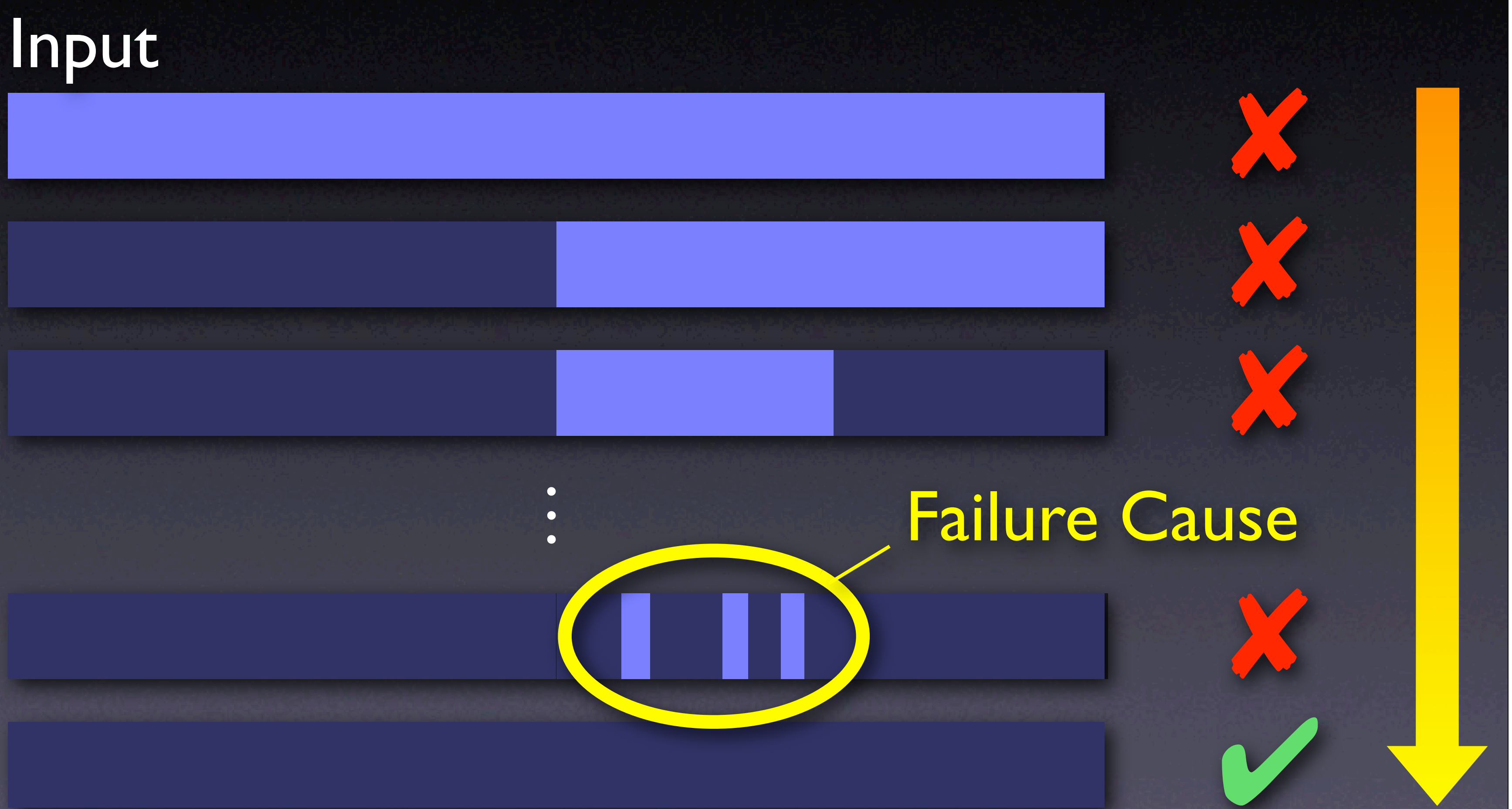
<SELECT NAME="priority" MULTIPLE SIZE=7>

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

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

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

Simplifying



Isolating Input

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

Difference narrowed down

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

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

Isolating Input

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

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

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

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

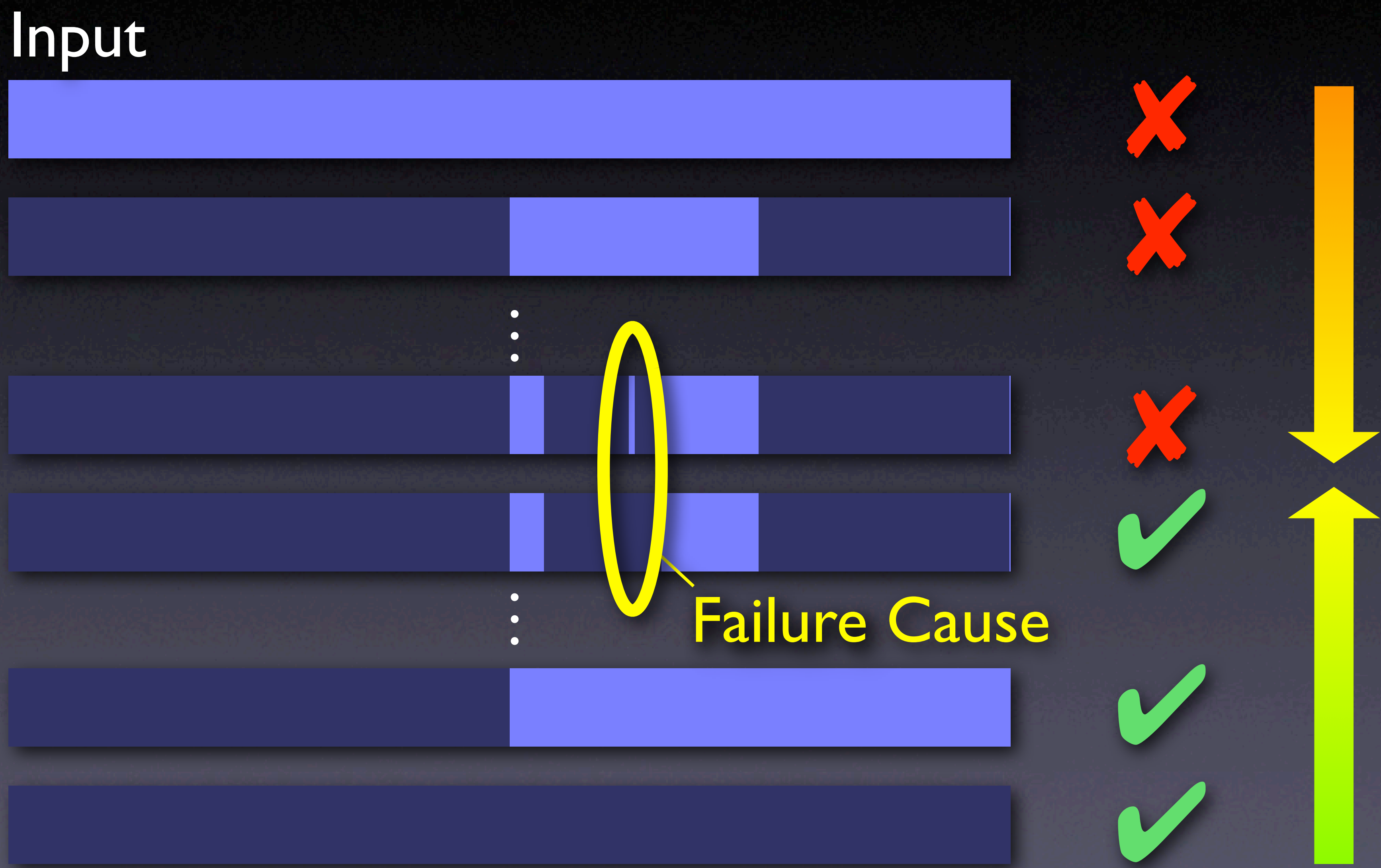
Failure Cause

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

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

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

Isolating



Configuration

Circumstance

$$\delta$$

All circumstances

$$C = \{\delta_1, \delta_2, \dots\}$$

Configuration $c \subseteq C$

$$c = \{\delta_1, \delta_2, \dots \delta_n\}$$

Tests

Testing function

$$test(c) \in \{✔, ✘, ?\}$$

Initial configurations

$$test(c_{✔}) = ✔$$
$$test(c_{✘}) = ✘$$

Minimal Difference

Goal: Subsets $c'_{✘}$ and $c'_{✔}$

$$\emptyset = c_{✔} \subseteq c'_{✔} \subset c'_{✘} \subseteq c_{✘}$$

Difference

$$\Delta = c'_{✘} \setminus c'_{✔}$$

Difference is 1-minimal

$$\forall \delta_i \in \Delta \cdot test(c'_{✔} \cup \{\delta_i\}) \neq ✔ \wedge test(c'_{✘} \setminus \{\delta_i\}) \neq ✘$$

Algorithm Sketch

- Extend *ddmin* such that it works on *two sets at a time* – $c'_{\times}$ and $c'_{\checkmark}$
- Compute subsets

$$\Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n = \Delta = c'_{\times} \setminus c'_{\checkmark}$$

- For each subset, test
  - the *addition* $c'_{\checkmark} \cup \Delta_i$
  - the *removal* $c'_{\times} \setminus \Delta_i$

Test Outcomes

	✘	✔
$test(c'_{✘} \setminus \Delta_i)$	$c'_{✘} := c'_{✘} \setminus \Delta_i$	$c'_{✔} := c'_{✘} \setminus \Delta_i$
$test(c'_{✔} \cup \Delta_i)$	$c'_{✘} := c'_{✔} \cup \Delta_i$	$c'_{✔} := c'_{✔} \cup \Delta_i$
otherwise	increase granularity	

most valuable outcomes

dd in a Nutshell

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

$$dd(c_{\checkmark}, c_{\times}) = dd'(c_{\checkmark}, c_{\times}, 2)$$

$$dd'(c'_{\checkmark}, c'_{\times}, n) =$$

$$\begin{cases} (c'_{\checkmark}, c'_{\times}) & \text{if } |\Delta| = 1 \\ dd'(c'_{\times} \setminus \Delta_i, c'_{\times}, 2) & \text{if } \exists i \in \{1..n\} \cdot test(c'_{\times} \setminus \Delta_i) = \checkmark \\ dd'(c'_{\checkmark}, c'_{\checkmark} \cup \Delta_i, 2) & \text{if } \exists i \in \{1..n\} \cdot test(c'_{\checkmark} \cup \Delta_i) = \times \\ dd'(c'_{\checkmark} \cup \Delta_i, c'_{\times}, \max(n-1, 2)) & \text{else if } \exists i \in \{1..n\} \cdot test(c'_{\checkmark} \cup \Delta_i) = \checkmark \\ dd'(c'_{\checkmark}, c'_{\times} \setminus \Delta_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1..n\} \cdot test(c'_{\times} \setminus \Delta_i) = \times \\ dd'(c'_{\checkmark}, c'_{\times}, \min(2n, |\Delta|)) & \text{else if } n < |\Delta| \text{ (“increase granularity”)} \\ (c'_{\checkmark}, c'_{\times}) & \text{otherwise} \end{cases}$$

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

Applications

Input	Code Changes	Schedules

Isolating Input

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

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

<SELECT NA ✓

<SELECT NA ✓

<SELECT NA ✓

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

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

Failure

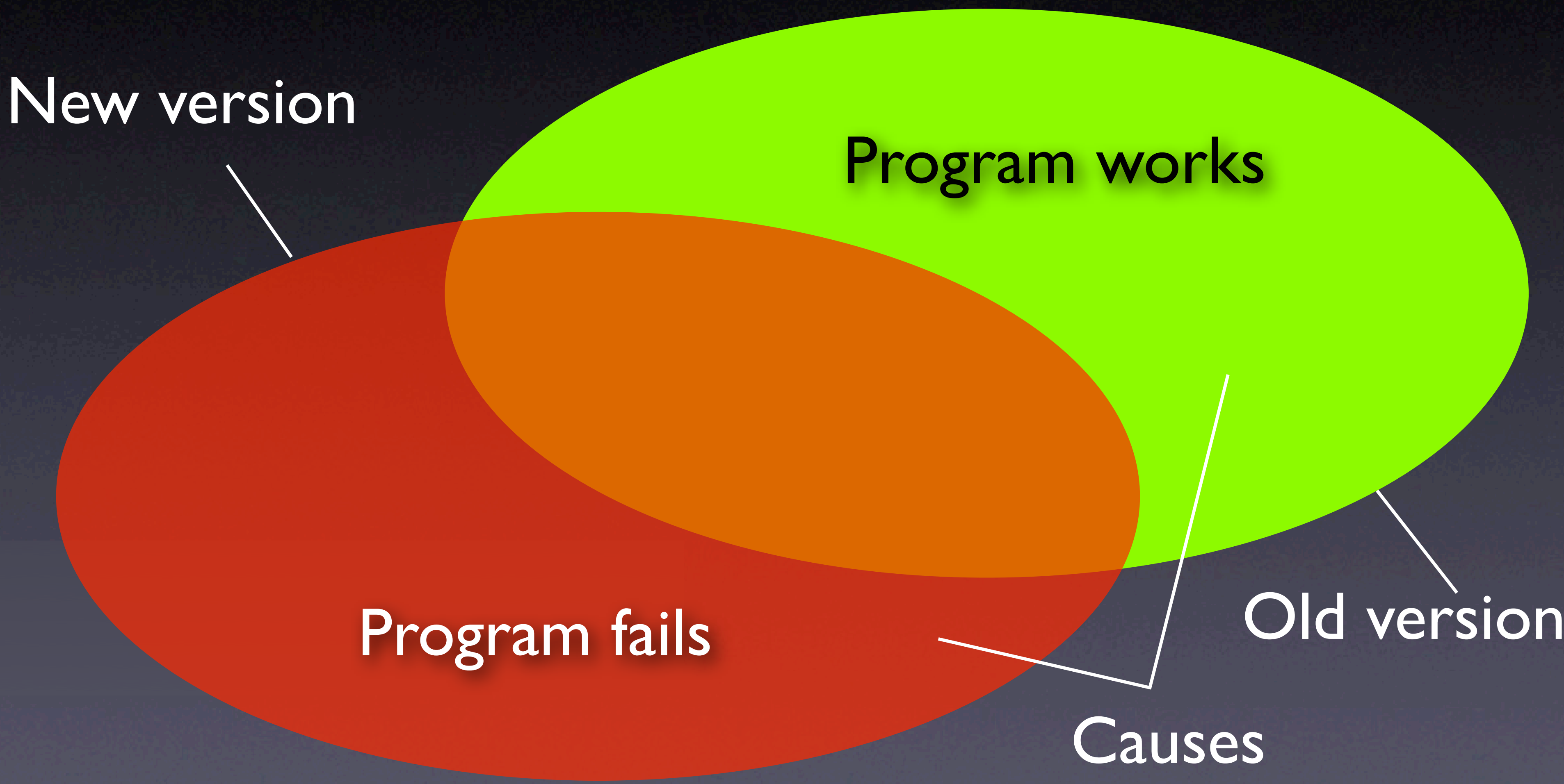
Isolation: 5 tests
Simplification: 48 tests

Code Changes

From: Brian Kahne <bkahne@ibmoto.com>
To: DDD Bug Report Address <bug-ddd@gnu.org>
Subject: Problem with DDD and GDB 4.17

When using DDD with GDB 4.16, the run command correctly uses any prior command-line arguments, or the value of "set args". However, when I switched to GDB 4.17, this no longer worked: If I entered a run command in the console window, the prior command-line options would be lost. [...]

Version Differences



What was Changed

```
$ diff -r gdb-4.16 gdb-4.17
diff -r gdb-4.16/COPYING gdb-4.17/COPYING
5c5
< 675 Mass Ave, Cambridge, MA 02139, USA
---
> 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
282c282
< Appendix: How to Apply These Terms to Your New Programs
---
> How to Apply These Terms to Your New Programs
```

…and so on for 178,200 lines (8,721 locations)

Challenges

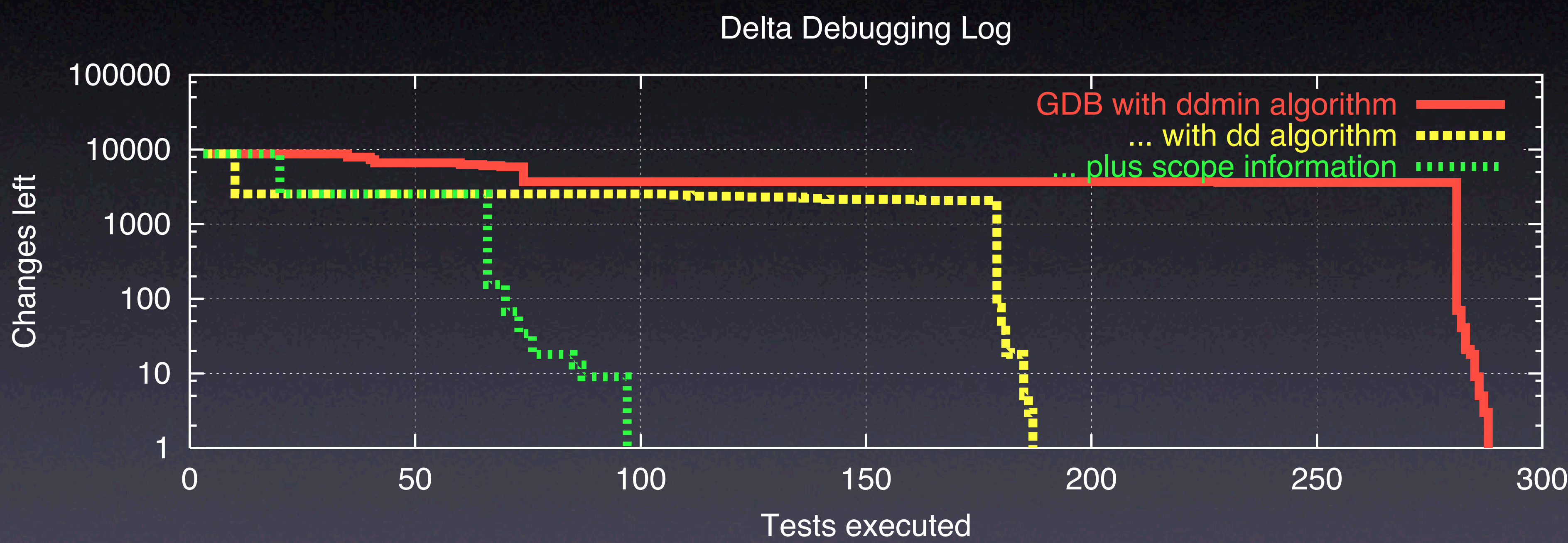
- Granularity – within some large change, only a few lines may be relevant
- Interference – some (later) changes rely on other (earlier) changes
- Inconsistency – some changes may have to be combined to produce testable code

Delta debugging handles all this

General Plan

- Decompose diff into changes per location (= 8,721 individual changes)
- Apply subset of changes, using PATCH
- Reconstruct GDB; build errors mean unresolved test outcome
- Test GDB and return outcome

Isolating Changes



- Result after 98 tests (= 1 hour)

The Failure Cause

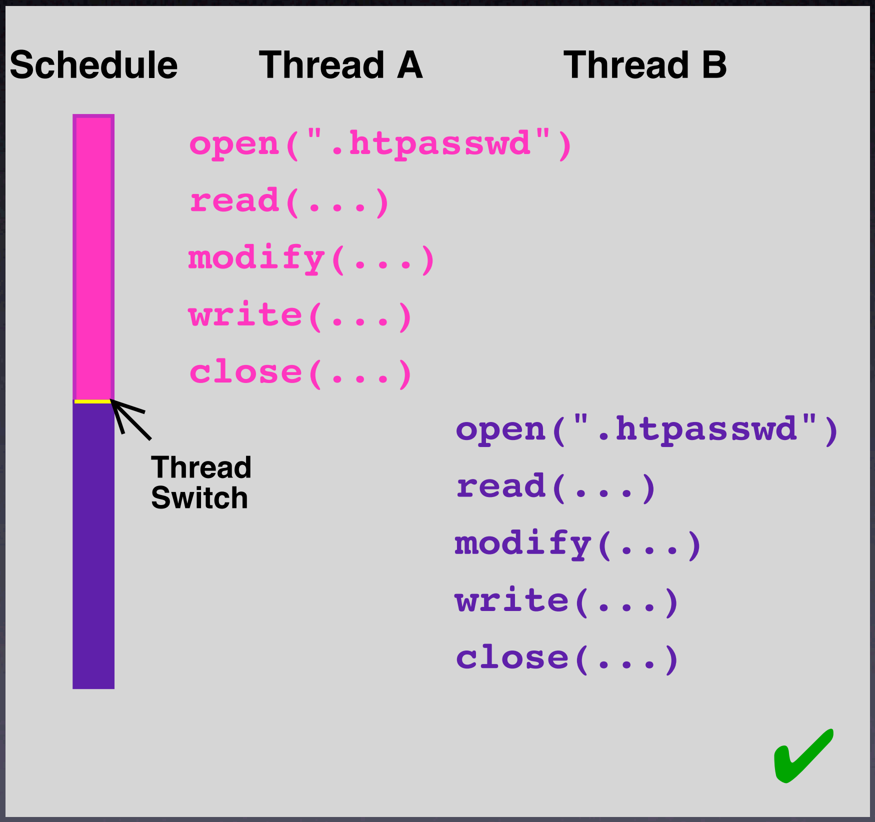
```
diff -r gdb-4.16/gdb/infcmd.c gdb-4.17/gdb/infcmd.c
1239c1278
< "Set arguments to give program being debugged when it is
started.\n
---
> "Set argument list to give program being debugged when
it is started.\n
```

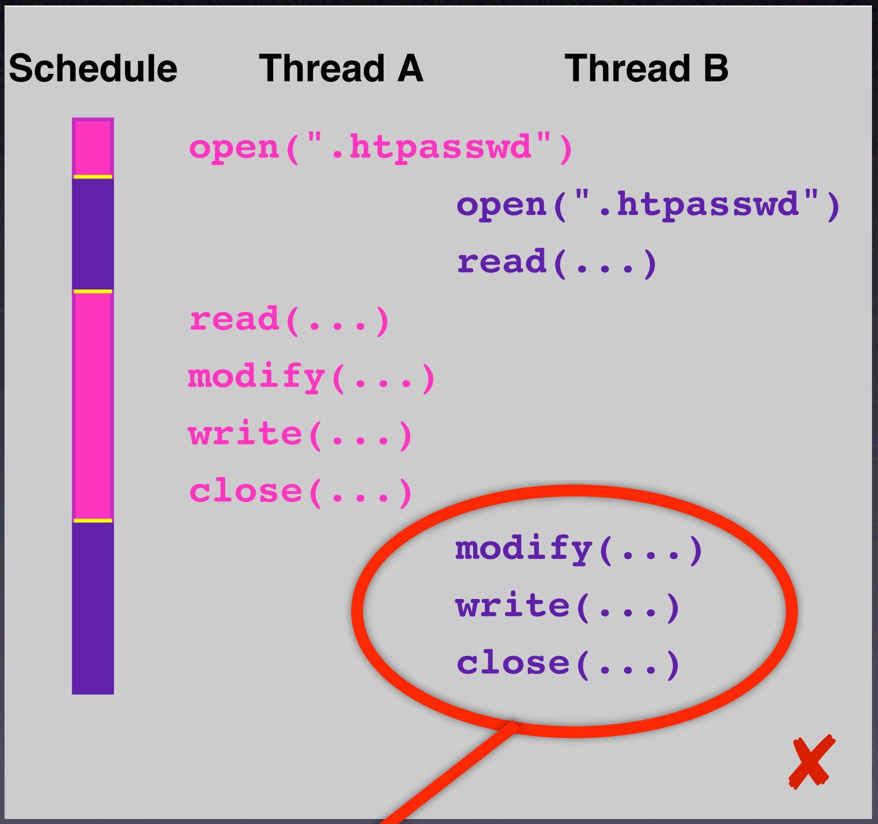
- Documentation becomes GDB output
- DDD expects Arguments,
  but GDB outputs Argument list

Optimizations

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

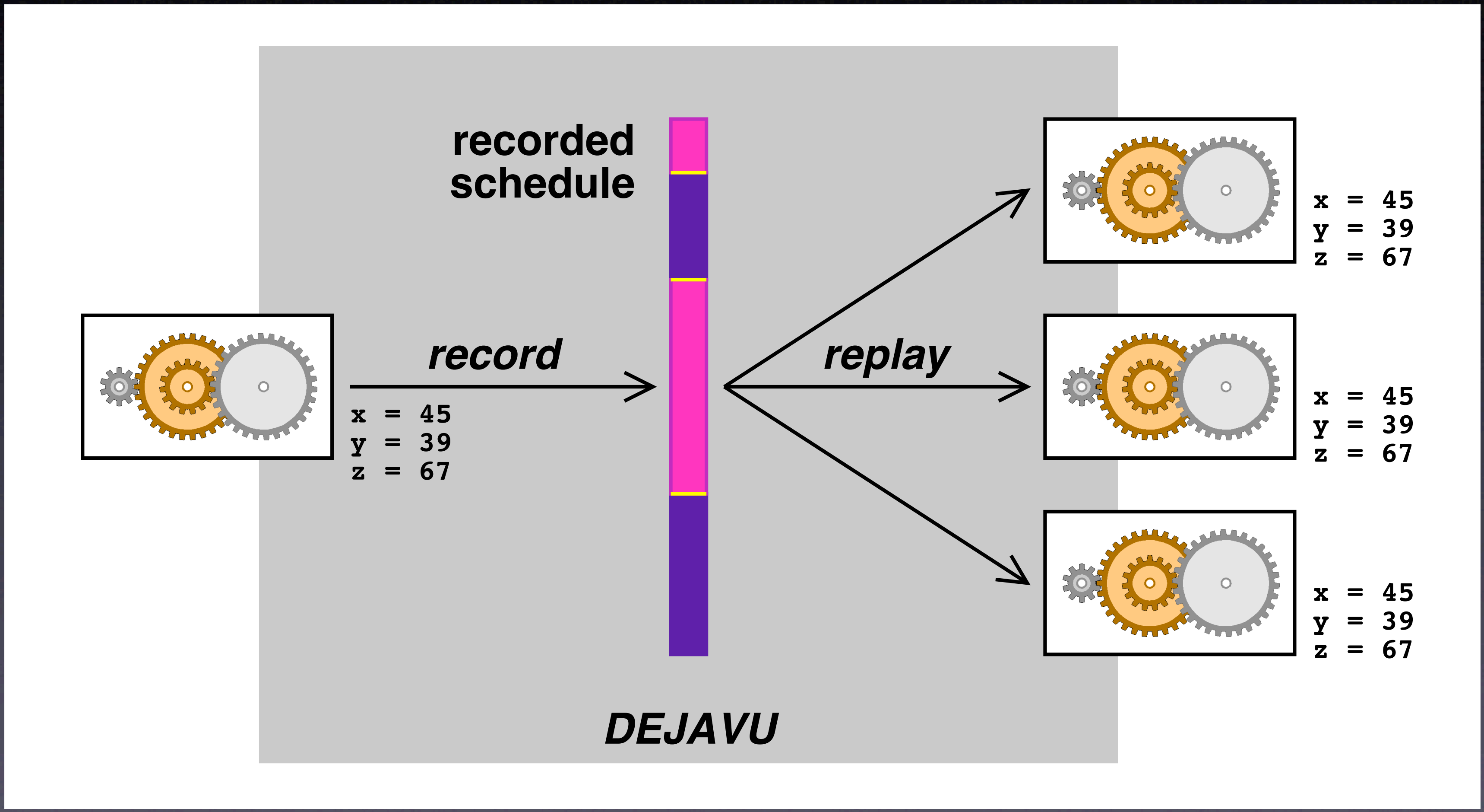
Thread Schedules



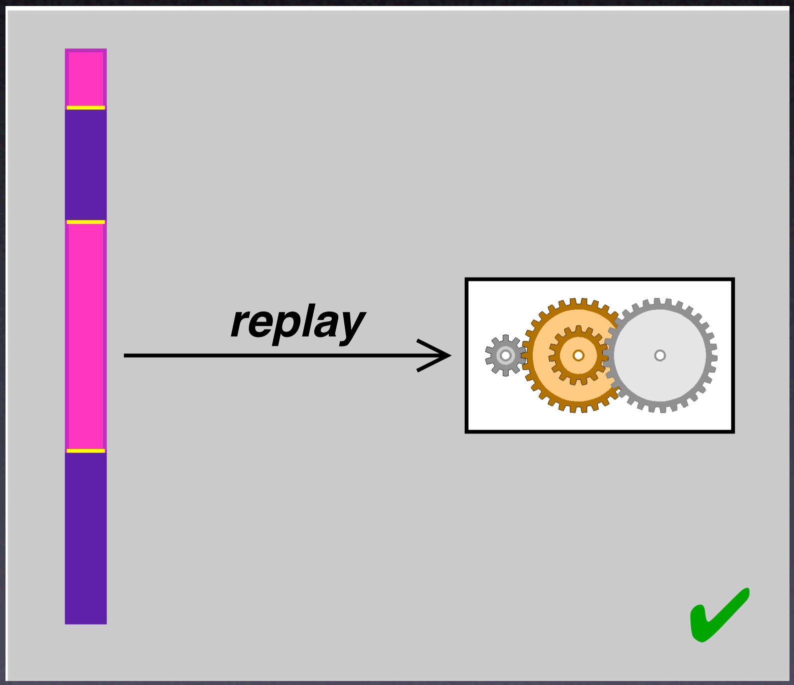


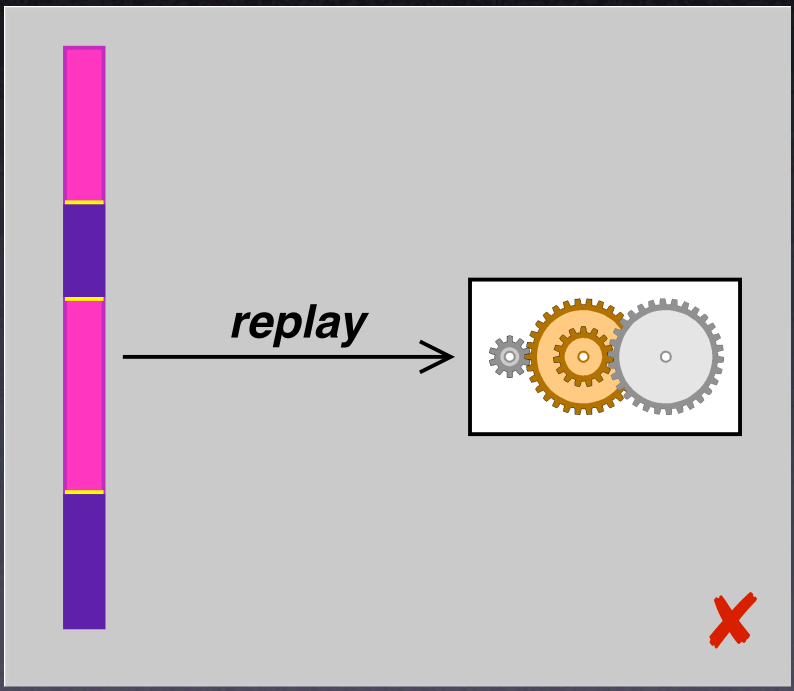
A's updates get lost!

Record + Replay



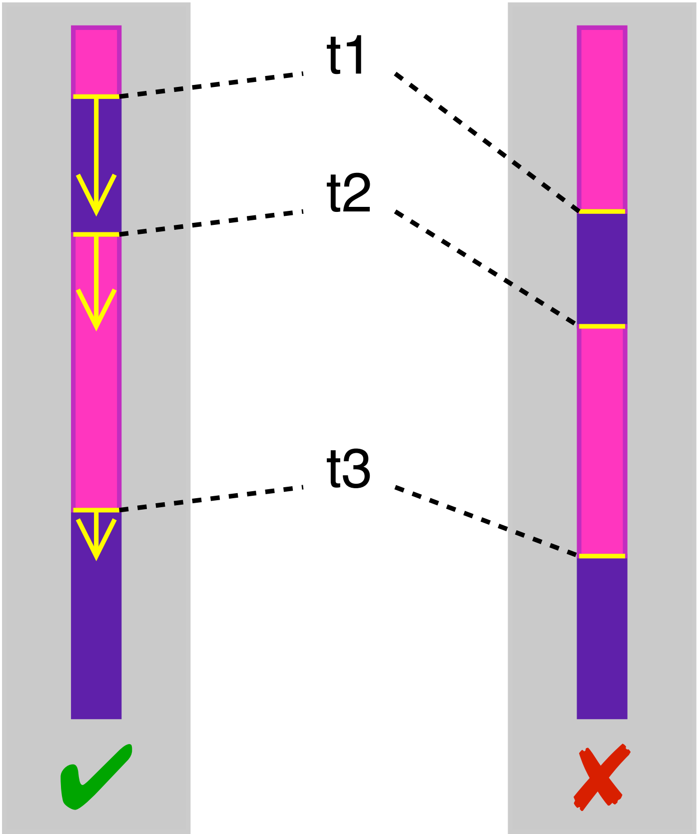
Schedules as Input





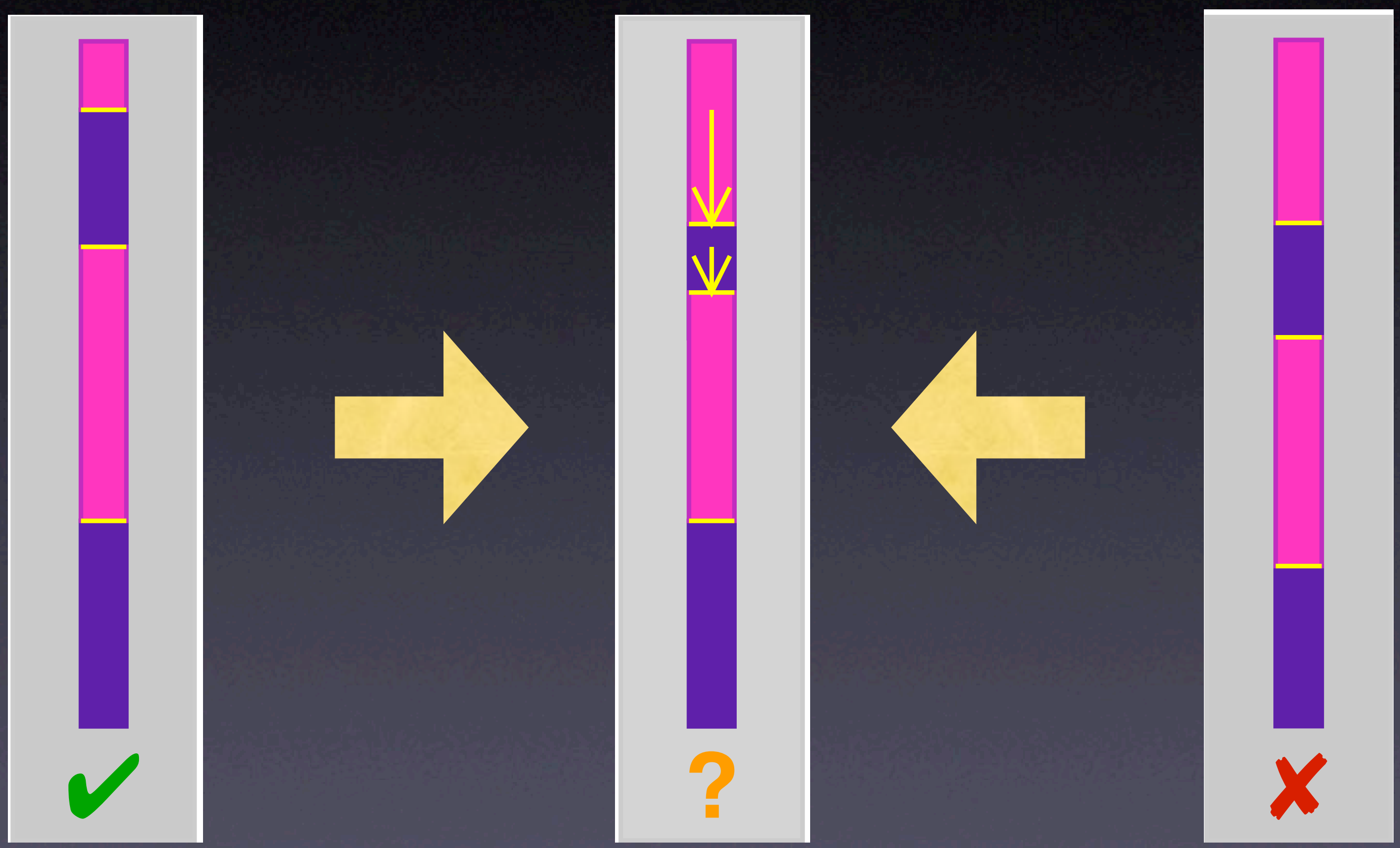
The schedule difference causes the failure!

Finding Differences

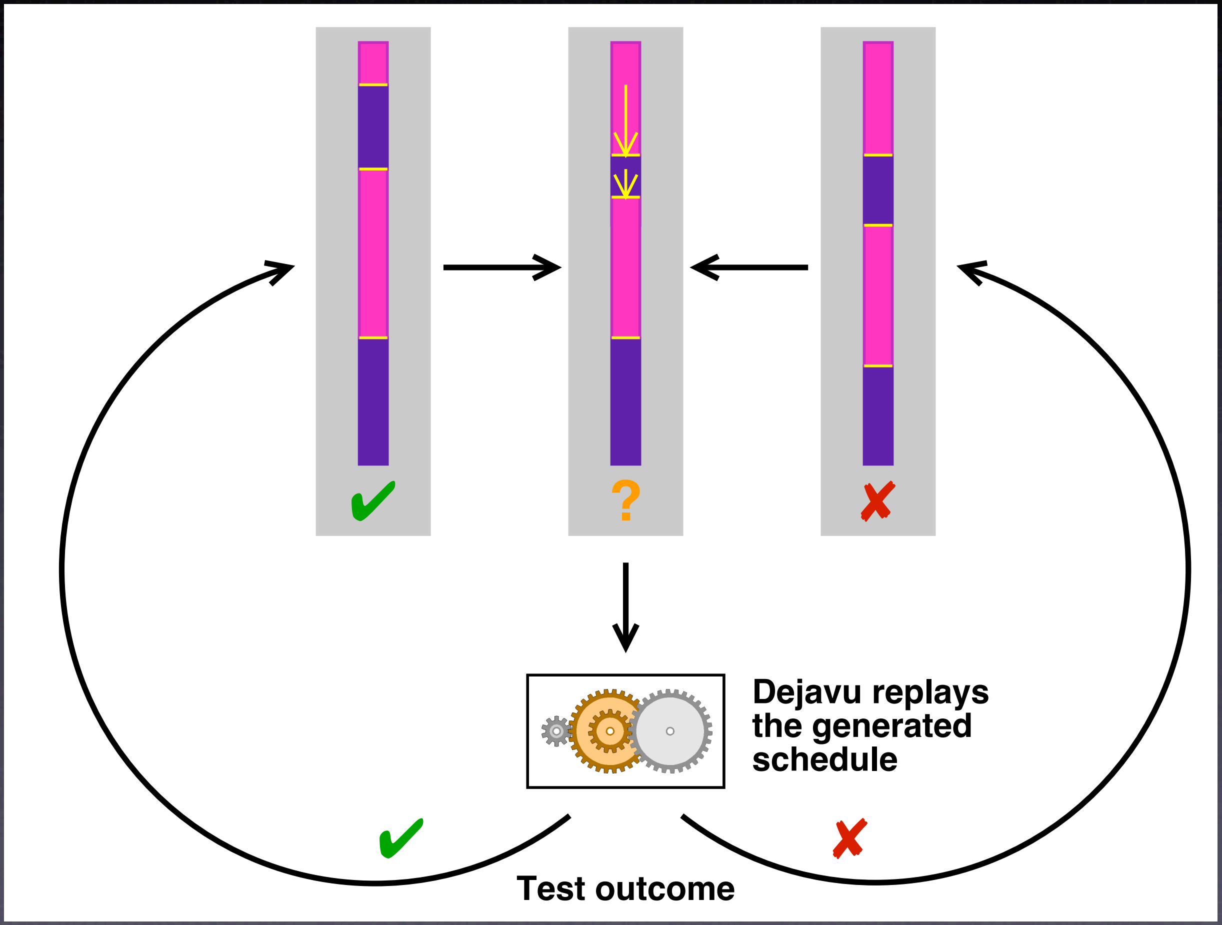


- We start with runs ✔ and ✘
- We determine the differences Δ_i between thread switches t_i:
  - t_1 occurs in ✔ at "time" 254
  - t_1 occurs in ✘ at "time" 278
  - The difference $\Delta_1 = |278 - 254|$ induces a *statement interval:* the code executed between "time" 254 and 278
  - Same applies to t_2, t_3, etc.

Isolating Differences



Isolating Differences

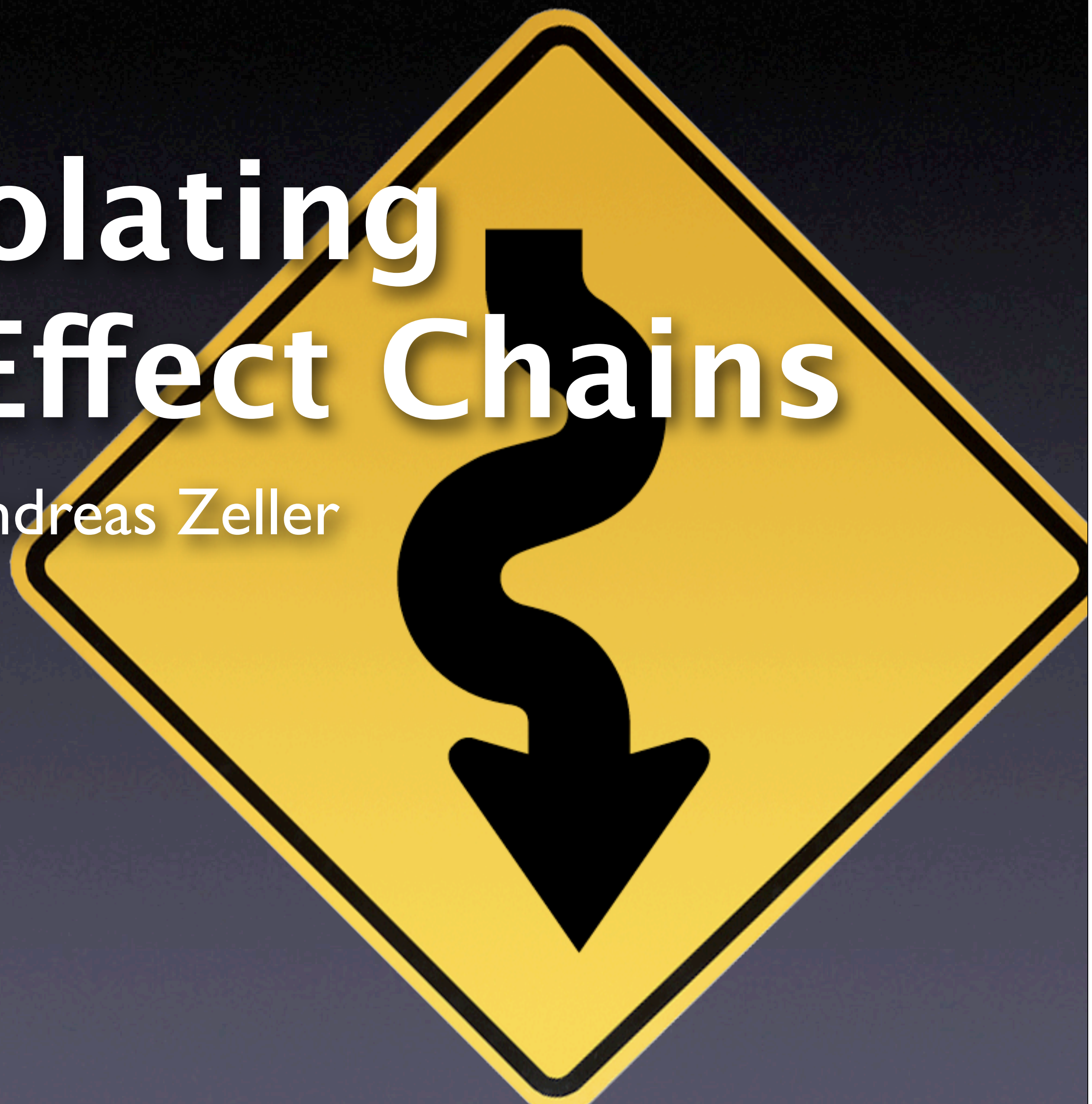


Example: Raytracer

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

Isolating Cause-Effect Chains

Andreas Zeller



bug.c

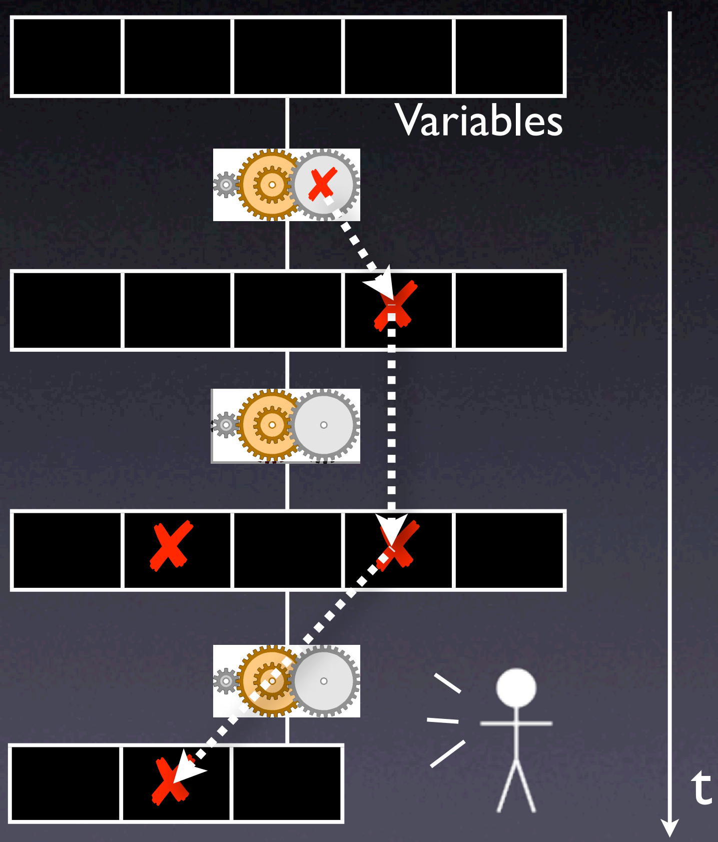
```
double bug(double z[], int n) {
    int i, j;

    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

What is the cause of this failure?

From Defect to Failure

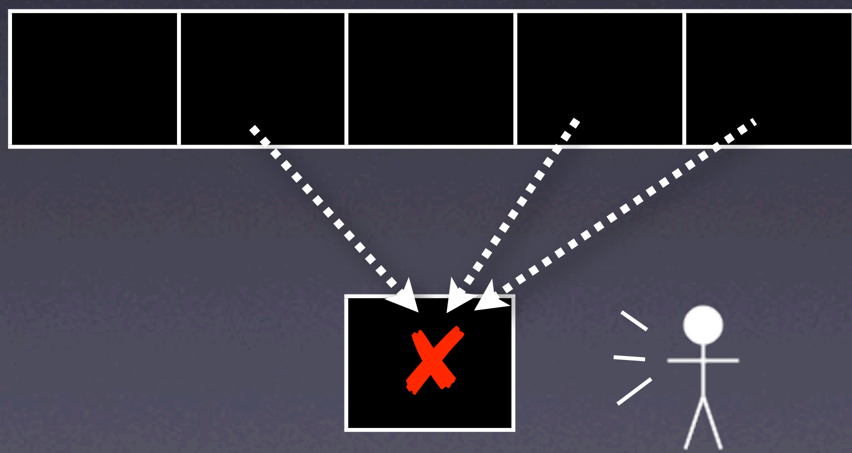
1. The programmer creates a *defect* – an error in the code.
2. When executed, the defect creates an *infection* – an error in the state.
3. The infection *propagates*.
4. The infection causes a *failure*.

This infection chain must be traced back – and broken.

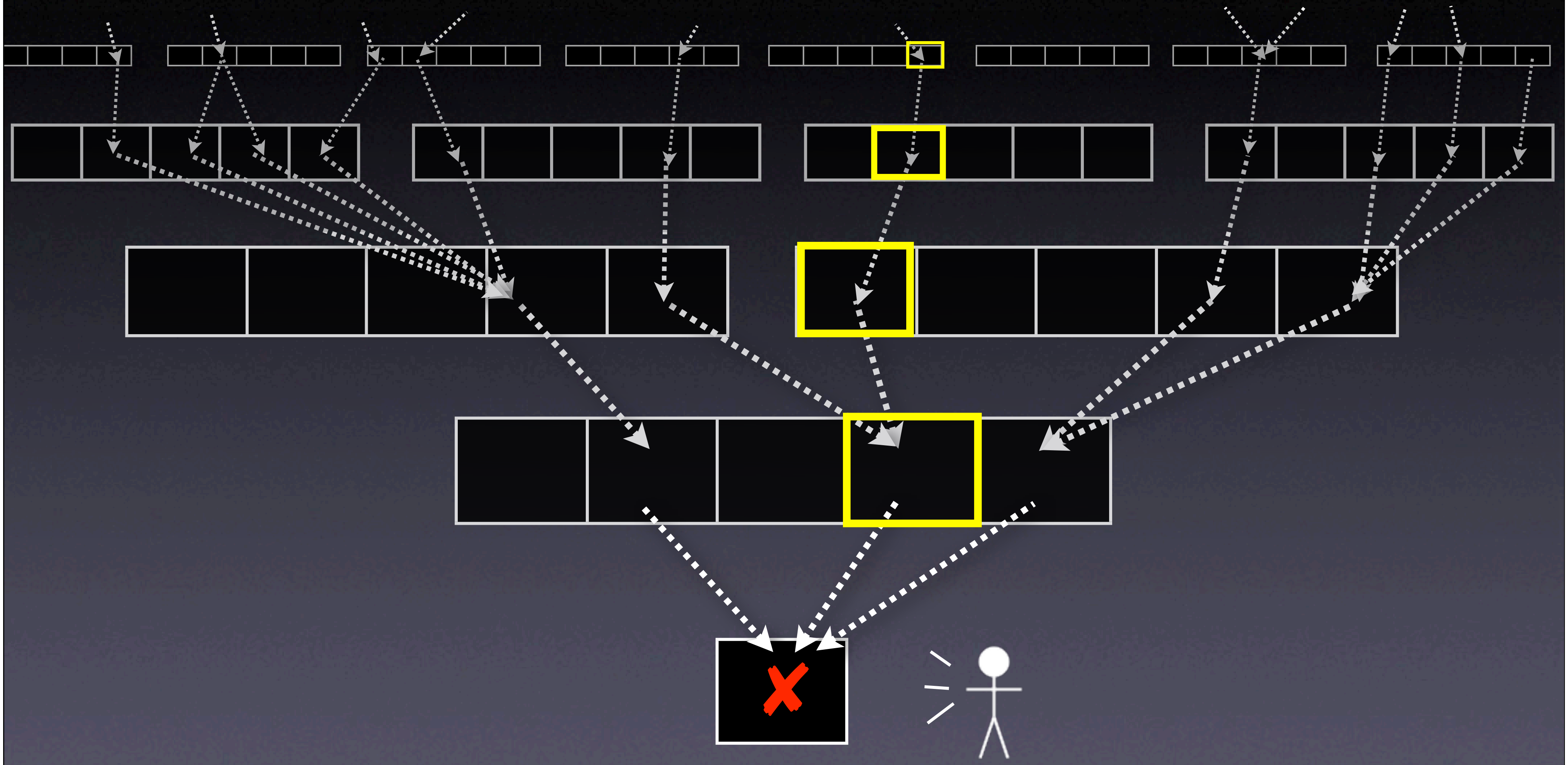


Tracing Infections

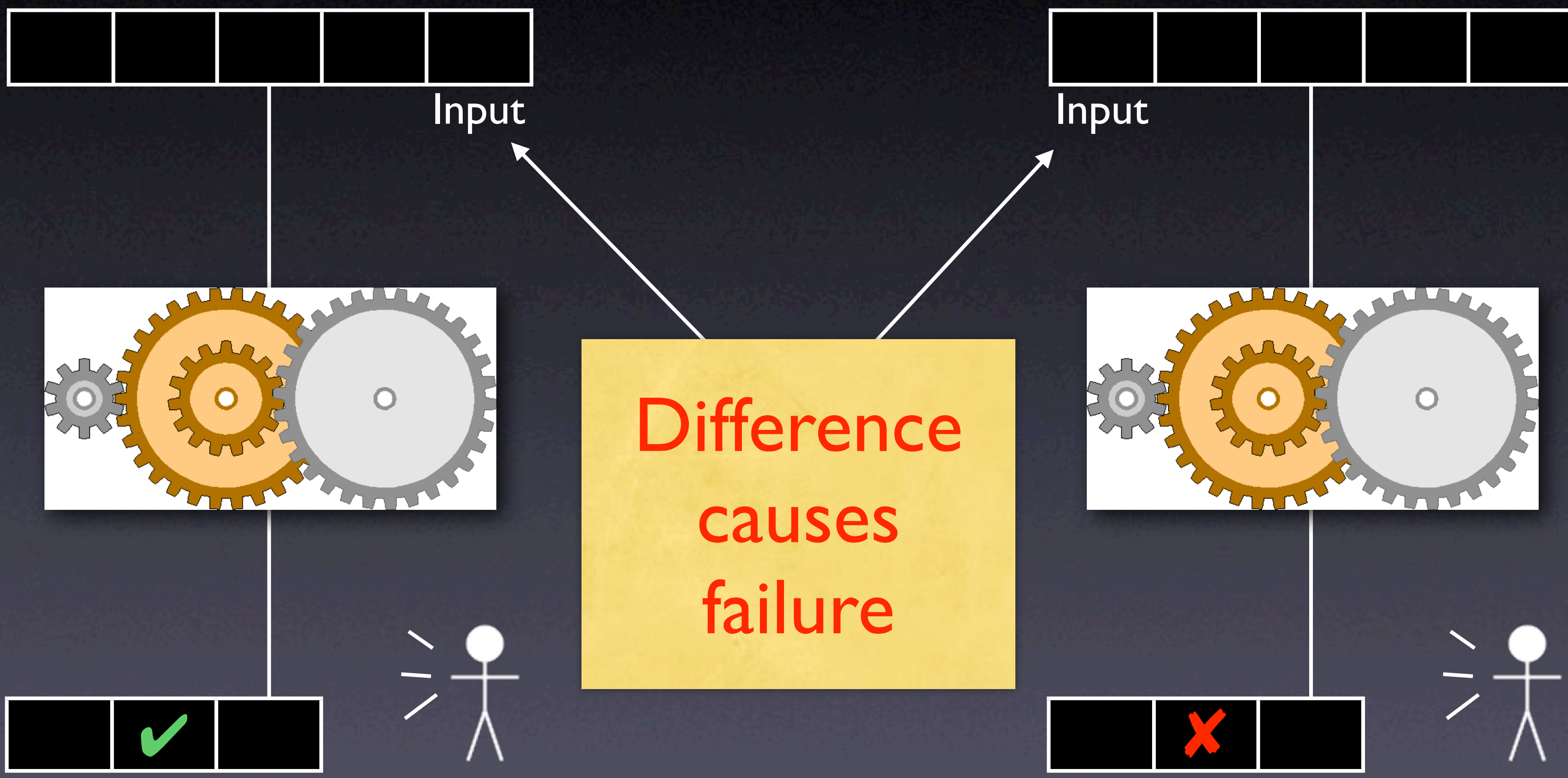
- For every infection, we must find the *earlier infection* that *causes* it.
- Program analysis tells us *possible causes*



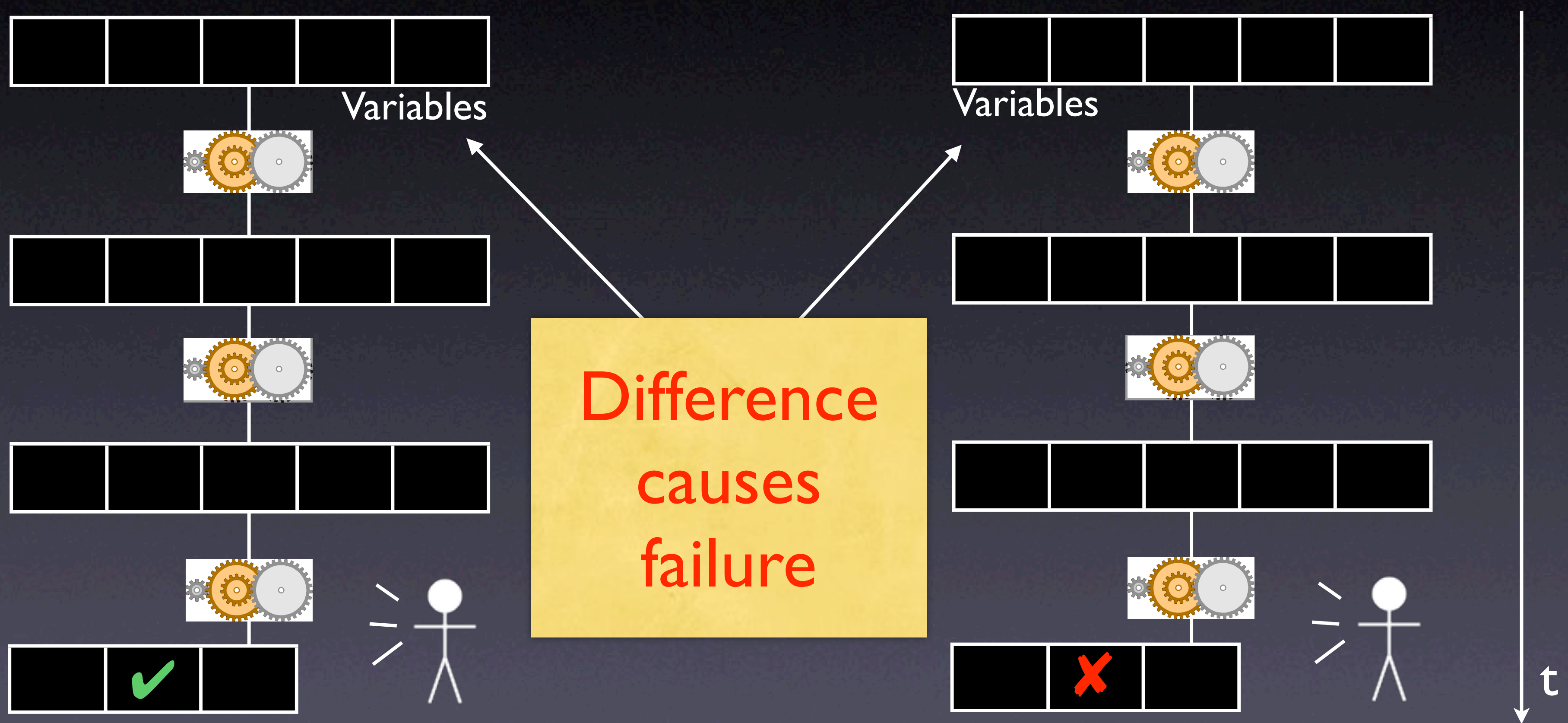
Tracing Infections



Isolating Input



Isolating States



Comparing States

- What is a program state, anyway?
- How can we compare states?
- How can we narrow down differences?

A Sample Program

```
$ sample 9 8 7
Output: 7 8 9

$ sample 11 14
Output: 0 11
```

Where is the defect
which causes this failure?

```c
int main(int argc, char *argv[])
{
    int *a;

    // Input array
    a = (int *)malloc((argc - 1) * sizeof(int));
    for (int i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    // Sort array
    shell_sort(a, argc);

    // Output array
    printf("Output: ");
    for (int i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

A sample state

- We can access the entire state via the debugger:

1. List all *base variables*
2. Expand all references…
3. …until a fixpoint is found

Sample States

Variable	Value	
	in $r_{✔}$	in $r_{✘}$
argc	4	5
argv[0]	"./sample"	"./sample"
argv[1]	"9"	"11"
argv[2]	"8"	"14"
argv[3]	"7"	0x0 (NIL)
i′	1073834752	1073834752
j	1074077312	1074077312
h	1961	1961
size	4	3

Variable	Value	
	in $r_{✔}$	in $r_{✘}$
i	3	2
a[0]	9	11
a[1]	8	14
a[2]	7	0
a[3]	1961	1961
a′[0]	9	11
a′[1]	8	14
a′[2]	7	0
a′[3]	1961	1961

at shell_sort()

Narrowing State Diffs

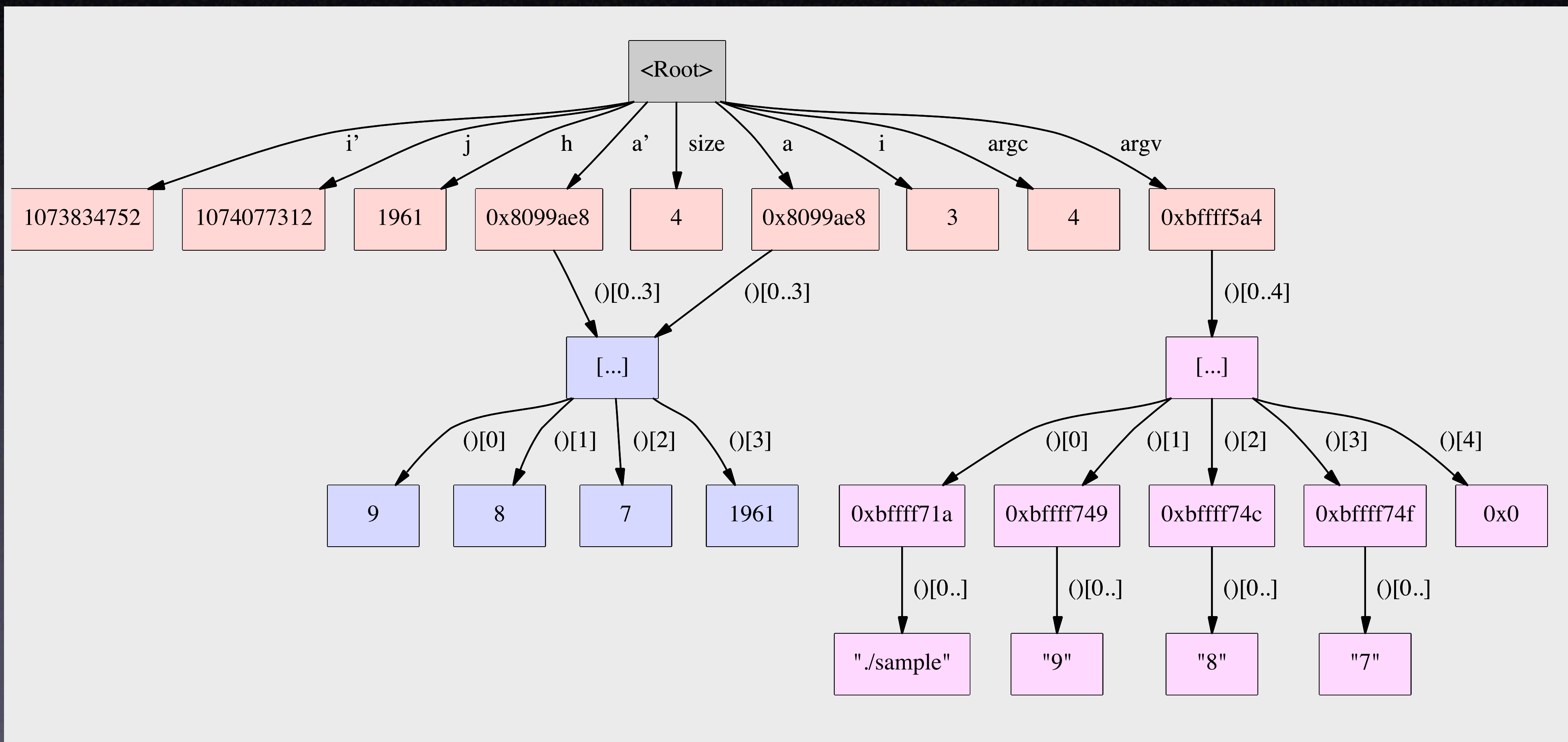
■ = δ is applied, □ = δ is *not* applied

#	$a'[0]$	$a[0]$	$a'[1]$	$a[1]$	$a'[2]$	$a[2]$	$argc$	$argv[1]$	$argv[2]$	$argv[3]$	i	$size$	Output	Test
1	□	□	□	□	□	□	□	□	□	□	□	□	7 8 9	✔
2	■	■	■	■	■	■	■	■	■	■	■	■	0 11	✘
3	■	■	■	■	■	■	□	□	□	□	□	□	0 11 14	✘
4	■	■	■	□	□	□	□	□	□	□	□	□	7 11 14	?
5	□	□	□	■	■	■	□	□	□	□	□	□	0 9 14	✘
6	□	□	□	■	□	□	□	□	□	□	□	□	7 9 14	?
7	□	□	□	□	■	■	□	□	□	□	□	□	0 8 9	✘
8	□	□	□	□	■	□	□	□	□	□	□	□	0 8 9	✘
Result					■									

Complex State

- Accessing the state as a *table* is not enough:
  - References are not handled
  - Aliases are not handled
- We need a *richer* representation

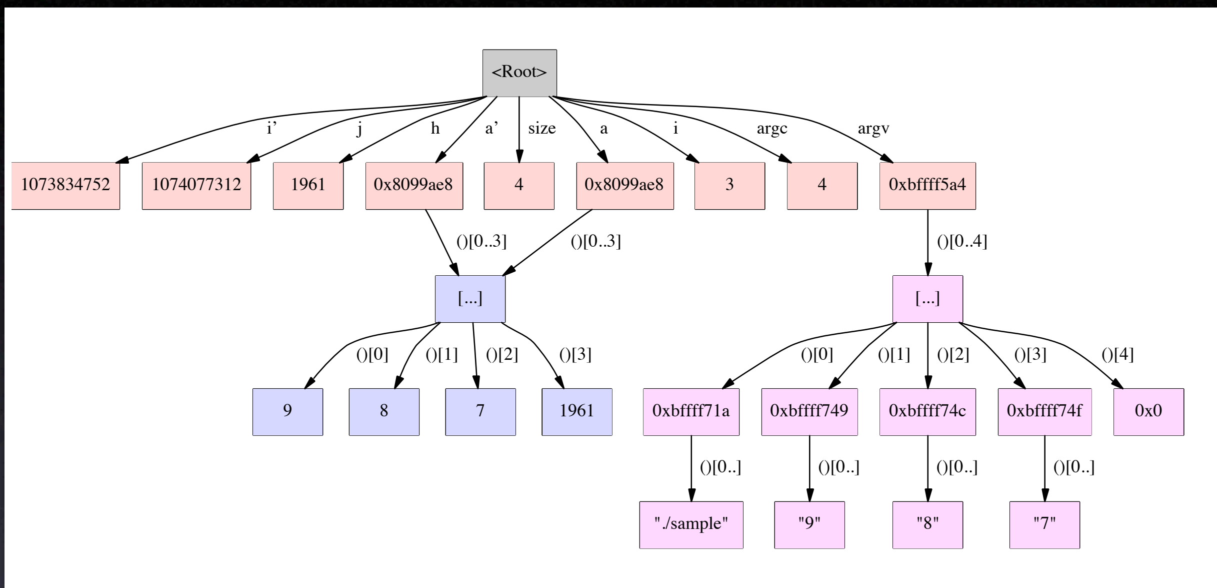
A Memory Graph



Unfolding Memory

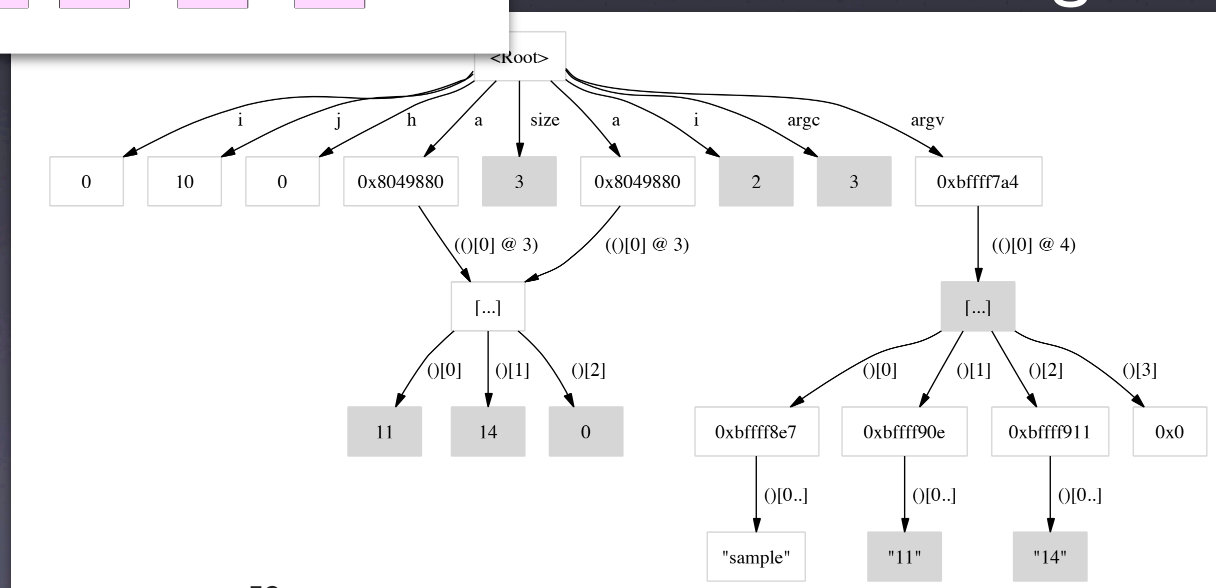
- Any variable: make new node
- Structures: unfold all members
- Arrays: unfold all elements
- Pointers: unfold object being pointed to
  - *Does p point to something? And how many?*

Comparing States



passing run

failing run

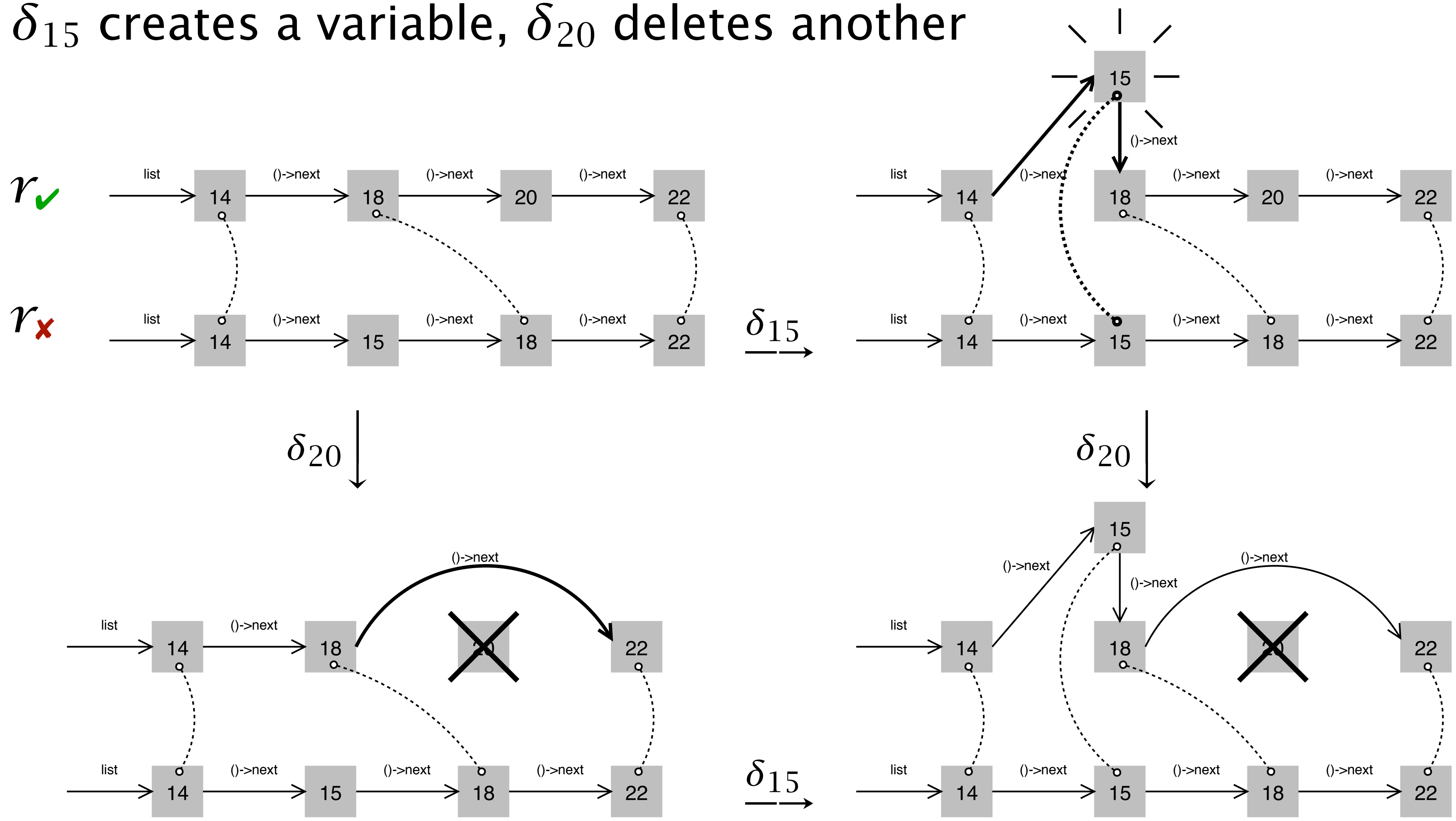


Comparing States

- Basic idea: *compute common subgraph*
- Any node that is not part of the common subgraph becomes a *difference*
- Applying a difference means to create or delete nodes – and adjust references
- All this is done within GDB

Applying Diffs

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



Results: GCC Transitions

#	Location	Cause transition to variable
0	⟨Start⟩	`argv[3]`
1	toplev.c:4755	`name`
2	toplev.c:2909	`dump_base_name`
3	c-lex.c:187	`finput→_IO_buf_base`
4	c-lex.c:1213	`nextchar`
5	c-lex.c:1213	`yyssa[41]`
6	c-typeck.c:3615	`yyssa[42]`
7	c-lex.c:1213	`last_insn→fld[1].rtx →fld[1].rtx→fld[3].rtx →fld[1].rtx.code`
8	c-decl.c:1213	`sequence_result[2] →fld[0].rtvec →elem[0].rtx→fld[1].rtx →fld[1].rtx→fld[1].rtx →fld[1].rtx→fld[1].rtx →fld[1].rtx→fld[1].rtx →fld[3].rtx→fld[1].rtx.code`
9	combine.c:4271	`x→fld[0].rtx→fld[0].rtx`

Concepts

- To isolate failure causes automatically, use
  - an *automated test case*
  - a means to *narrow down the difference*
  - a *strategy* for proceeding.
- One possible strategy is Delta Debugging.

Concepts (2)

- Delta Debugging can isolate failure causes
  - in the (general) *input*
  - in the *version history*
  - in *thread schedules*
  - *in program states*
- Every such cause implies a *fix* – but not necessarily a correction.

Announcement

Dear students,

I updated the lecture schedule. Most notable changes are

- I removed (R) signs from several papers making them as optional.
  Reps' et al.'s profiling paper for 4/8,
  Lanza et al.'s paper on metrics and visualization for 4/20,
  Boshernitsan's paper on source transformation for 4/29
  If you are signed up for these papers, you are still scheduled to present. However, I won't discuss these papers in depth during my lecture.

- I switched the order between Lanza et al.'s and Murphy et al.'s paper.

- For next monday, I will talk about using delta-debugging for isolating cause-effect chain. It's likely that we will have more discussion on regression testing on next wednesday instead. If you are signed up for presenting Orso et al's paper, you are still on for monday.

Thanks!
Miryung