Lecture 20

Delta Debugging
Regression Testing

Today's Agenda

- Regression Testing
  - Presentation by Divya (advocate)
  - Presentation by David (skeptic)
- Delta Debugging:
  - Understanding its problem characterization one more time!!!
  - Quiz answers
  - Isolating Cause and Effect Chain

Delta Debugging Problem (1)

Circumstance

$$\delta$$

All circumstances

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

Configuration $c \subseteq C$

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

Delta Debugging Problem (2)

Testing function

$$test(c) \in \{✔, ✘, ?\}$$

Failure-inducing configuration

$$test(c_{✘}) = ✘$$

Relevant configuration $c'_{✘} \subseteq c_{✘}$

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

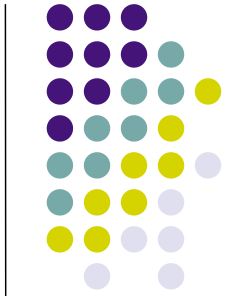
Mappings to DD Framework

	Circumstances (δ)	Configuration (c)	Testing Function test(c)		
			✔	✖	?
Simplifying Inputs (Zeller, FSE 99)	A set of inputs	A subset of the inputs	Running a test function on the input c		
Quiz					
Identifying Failure Inducing Changes (Zeller, FSE 99)	Changes	A subset of the changes	Running a test function on a base program + changes (c)		
DDD 3.1.2 case study	344 textual deltas between 3.1.1 and 3.1.2	DDD 3.1.1 and deltas up to a particular date	Invoking DDD with the name of a non-existing file		
			no core dump	core dump	can't compile DDD
GDB 4.17					

	Circumstances (δ)	Configuration (c)	Testing Function test(c)		
			✔	✖	?
Simplifying Inputs (Zeller, FSE 99)	A set of inputs	A subset of the inputs	Running the test code on the input c		
Quiz	A sequence of values in an array	A subsequence of values in an array	testSort that takes a sequence		
Identifying Failure Inducing Changes (Zeller, FSE 99)	Changes	A subset of the changes	Running the test code on a base program + changes (c)		
DDD 3.1.2 case study	344 textual deltas between 3.1.1 and 3.1.2	DDD 3.1.1 and deltas up to a particular date	Invoking DDD with the name of a non-existing file		
			no core dump	core dump	can't compile DDD
GDB 4.17	8721 textual deltas	GDB 4.16 and a subset of deltas	Passing arguments in DDD front-end to GDB		
			Arguments passed	Arguments not passed	Can't compile GDB

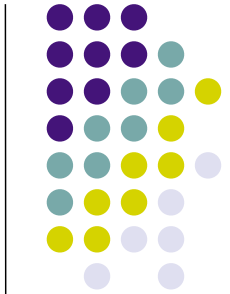
	Circumstances (δ)	Configuration (c)	Testing Function test(c)		
			✔	✖	?
Identifying Failure Inducing Thread Schedule (ISSTA 2002)	A set of context switch events	A subset of the events	Run a program with the schedule		
Identifying Cause Effect Chain (FSE 2002)	A set of (variable, value) pairs	A subset of (variable, value) pairs	Resume the debugger with the modified (variable, value) pairs		
GCC	a set of (variable and value) pairs at a particular debugger breakpoint	a subset of (variable, value) pairs	Running GCC on the fail.c as input		
			no crash	crash	
Locating Failure Causes (ICSE 2005)	A set of debug breakpoints that include failure-inducing program states	A subset of debug breakpoints that include failure-inducing program states	Resume the debugger with the modified (variable, value) pairs		

ddmin algorithm



```
def ddmin(circumstances, n):
    while len(circumstances) >= 2:
        subsets = split(circumstances, n)
        some_complement_return_false = 0
        for subset in subsets:
            complement = listminus(circumstances, subset)
            if testSort(complement) == False:
                circumstances = complement
                n = max(n - 1, 2)
                some_complement_return_false = 1
                break
            if not some_complement_return_false:
                if n == len(circumstances):
                    break
            n = min(n * 2, len(circumstances))
    return circumstances
```

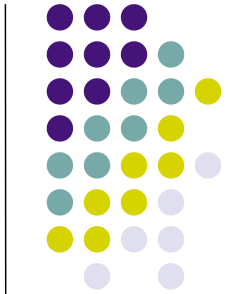
ddmin algorithm



Input: [0, 1, 2, 3, 5, 4, 5, 6]

Step	n	circumstances	complement	testSort (complement)
1	2	[0, 1, 2, 3, 5, 4, 5, 6]	[5,4,5,6]	false

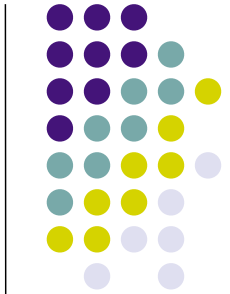
ddmin algorithm



Input: [0, 1, 2, 3, 5, 4, 5, 6]

Step	n	circumstances	complement	testSort (complement)
1	2	[0, 1, 2, 3, 5, 4, 5, 6]	[5,4,5,6]	false
2	2	[5,4,5,6]	[5,6]	true

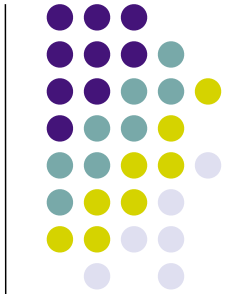
ddmin algorithm



Input: [0, 1, 2, 3, 5, 4, 5, 6]

Step	n	circumstances	complement	testSort (complement)
1	2	[0, 1, 2, 3, 5, 4, 5, 6]	[5,4,5,6]	false
2	2	[5,4,5,6]	[5,6]	true
3	2	[5,4,5,6]	[5,4]	false

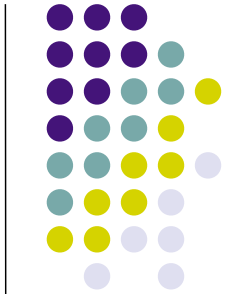
ddmin algorithm



Input: [0, 1, 2, 3, 5, 4, 5, 6]

Step	n	circumstances	complement	testSort (complement)
1	2	[0, 1, 2, 3, 5, 4, 5, 6]	[5,4,5,6]	false
2	2	[5,4,5,6]	[5,6]	true
3	2	[5,4,5,6]	[5,4]	false
4	2	[5,4]	[4]	true

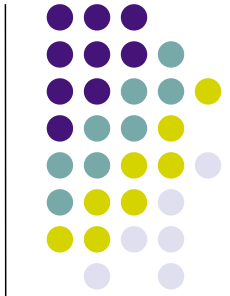
ddmin algorithm



Input: [0, 1, 2, 3, 5, 4, 5, 6]

Step	n	circumstances	complement	testSort (complement)
1	2	[0, 1, 2, 3, 5, 4, 5, 6]	[5,4,5,6]	false
2	2	[5,4,5,6]	[5,6]	true
3	2	[5,4,5,6]	[5,4]	false
4	2	[5,4]	[4]	true
5	2	[5,4]	[5]	true

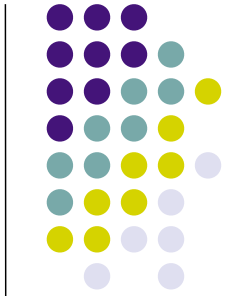
ddmin algorithm



Input: [3, 5, 7, 6, 8, 9, 13, 11]

Step	n	circumstances	complement	testSort (complement)
1	2	[3,5,7,6,8,9,13,11]	[8,9,13,11]	false

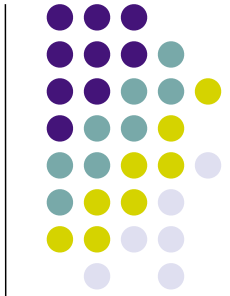
ddmin algorithm



Input: [3, 5, 7, 6, 8, 9, 13, 11]

Step	n	circumstances	complement	testSort (complement)
1	2	[3,5,7,6,8,9,13,11]	[8,9,13,11]	false
2	2	[8,9,13,11]	[13,11]	false

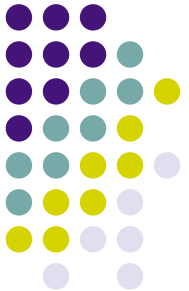
ddmin algorithm



Input: [3, 5, 7, 6, 8, 9, 13, 11]

Step	n	circumstances	complement	testSort (complement)
1	2	[3,5,7,6,8,9,13,11]	[8,9,13,11]	false
2	2	[8,9,13,11]	[13,11]	false
3	2	[13,11]	[11]	true

ddmin algorithm

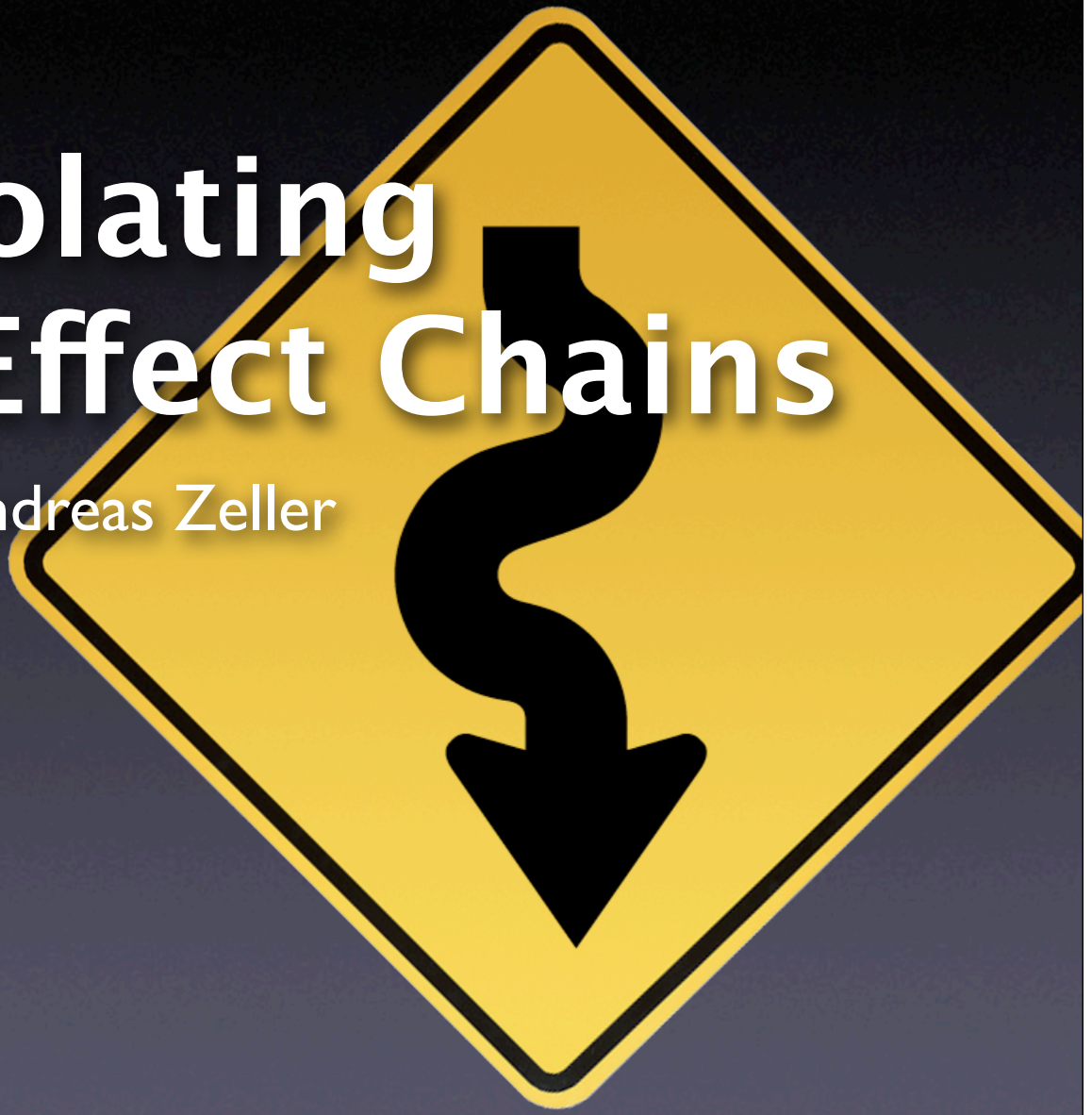


Input: [3, 5, 7, 6, 8, 9, 13, 11]

Step	n	circumstances	complement	testSort (complement)
1	2	[3,5,7,6,8,9,13,11]	[8,9,13,11]	false
2	2	[8,9,13,11]	[13,11]	false
3	2	[13,11]	[11]	true
4	2	[13,11]	[13]	true

Isolating Cause–Effect Chains

Andreas Zeller



bug.c

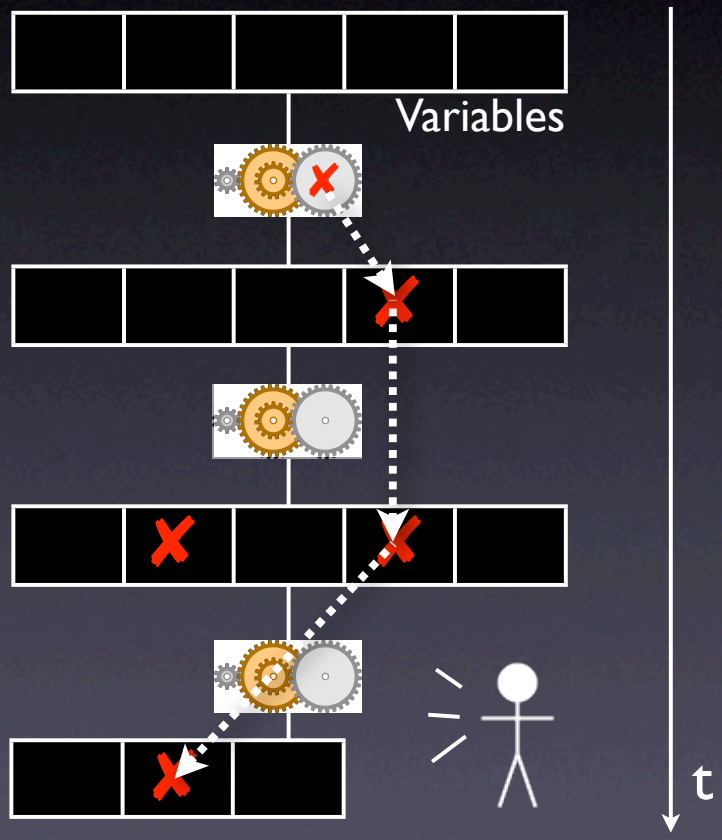
```
double bug(double z[], int n) {
    int i, j;

    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

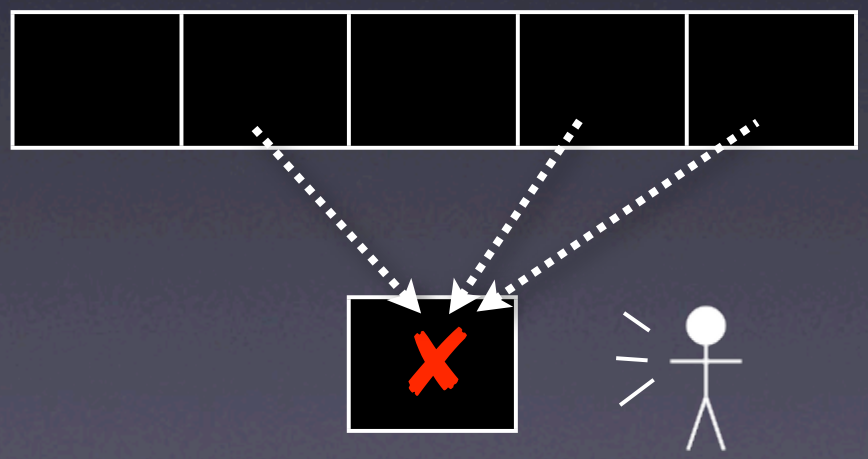
1. The programmer creates a *defect* – an error in the code.
2. When executed, the defect creates an *infection* – an error in the state.
3. The infection *propagates*.
4. The infection causes a *failure*.

This infection chain must be traced back – and broken.

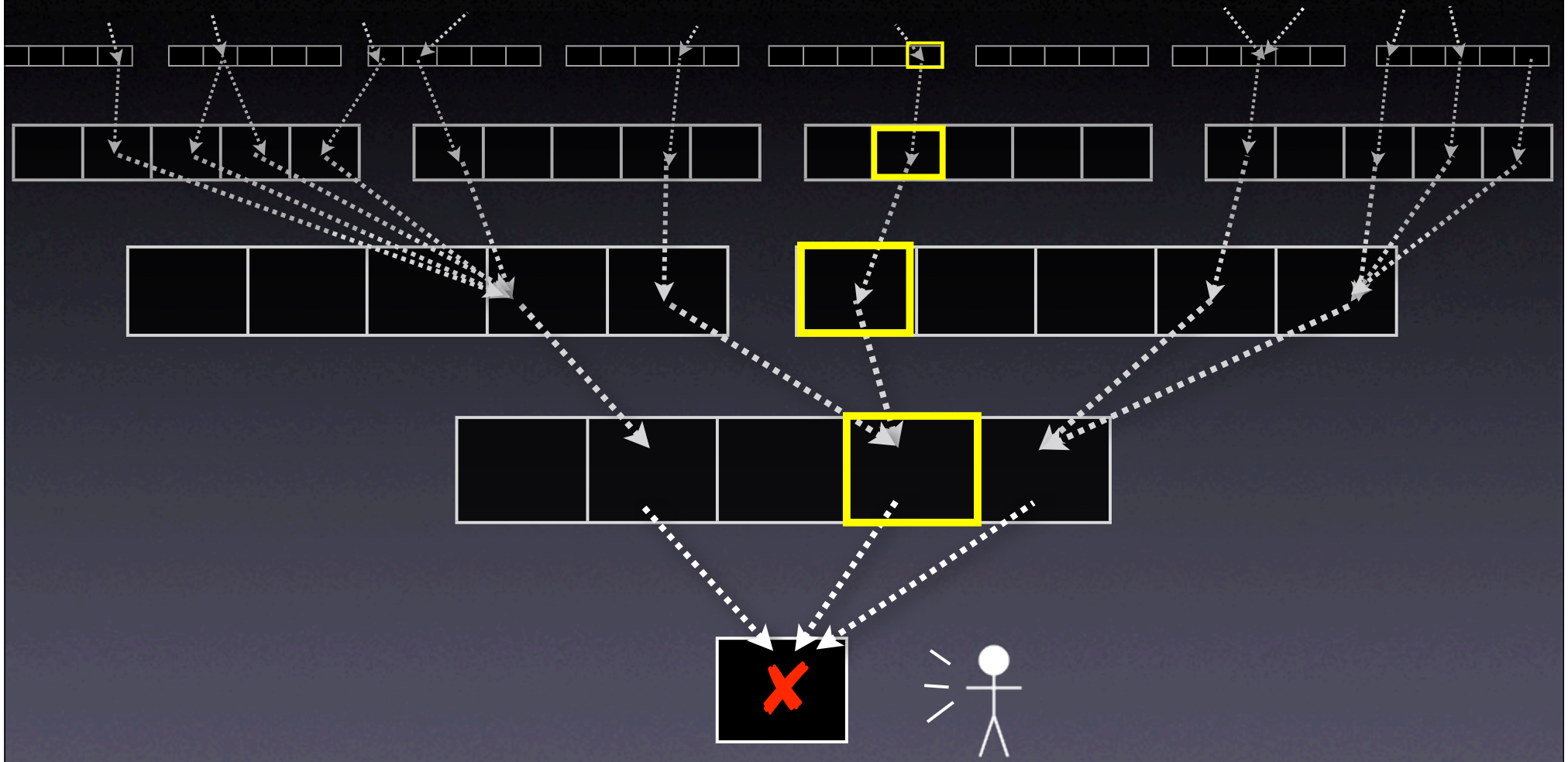


Tracing Infections

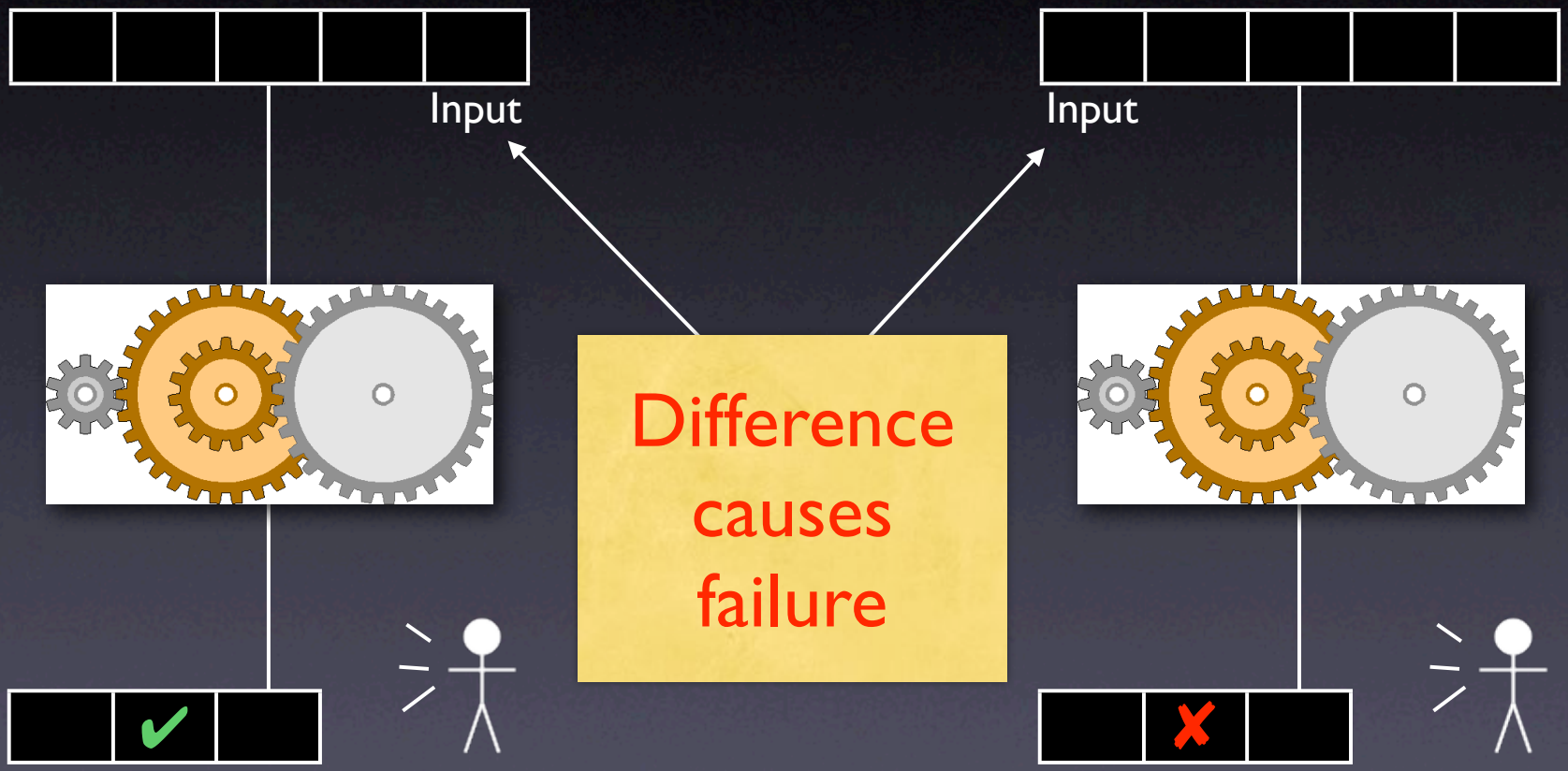
- For every infection, we must find the *earlier infection* that *causes* it.
- Program analysis tells us *possible causes*



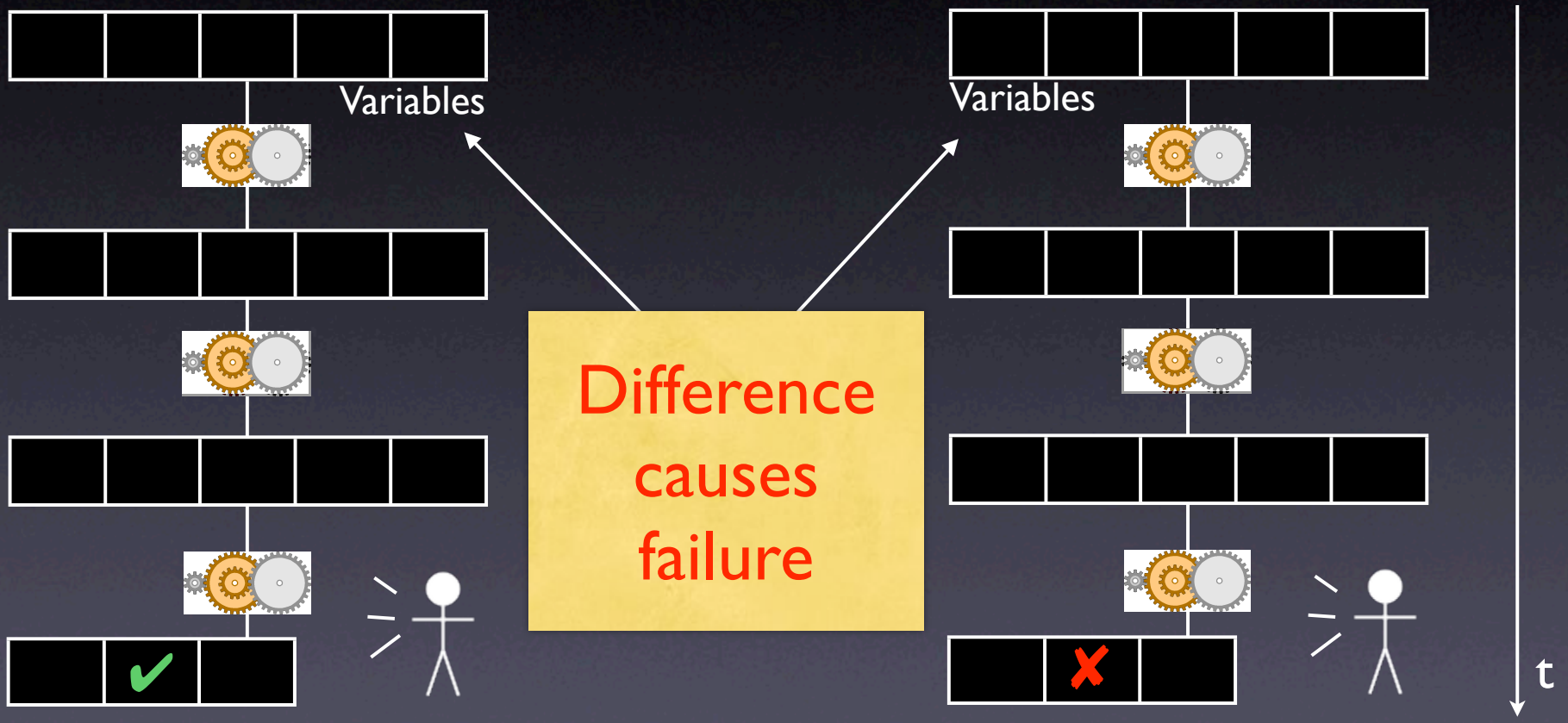
Tracing Infections



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

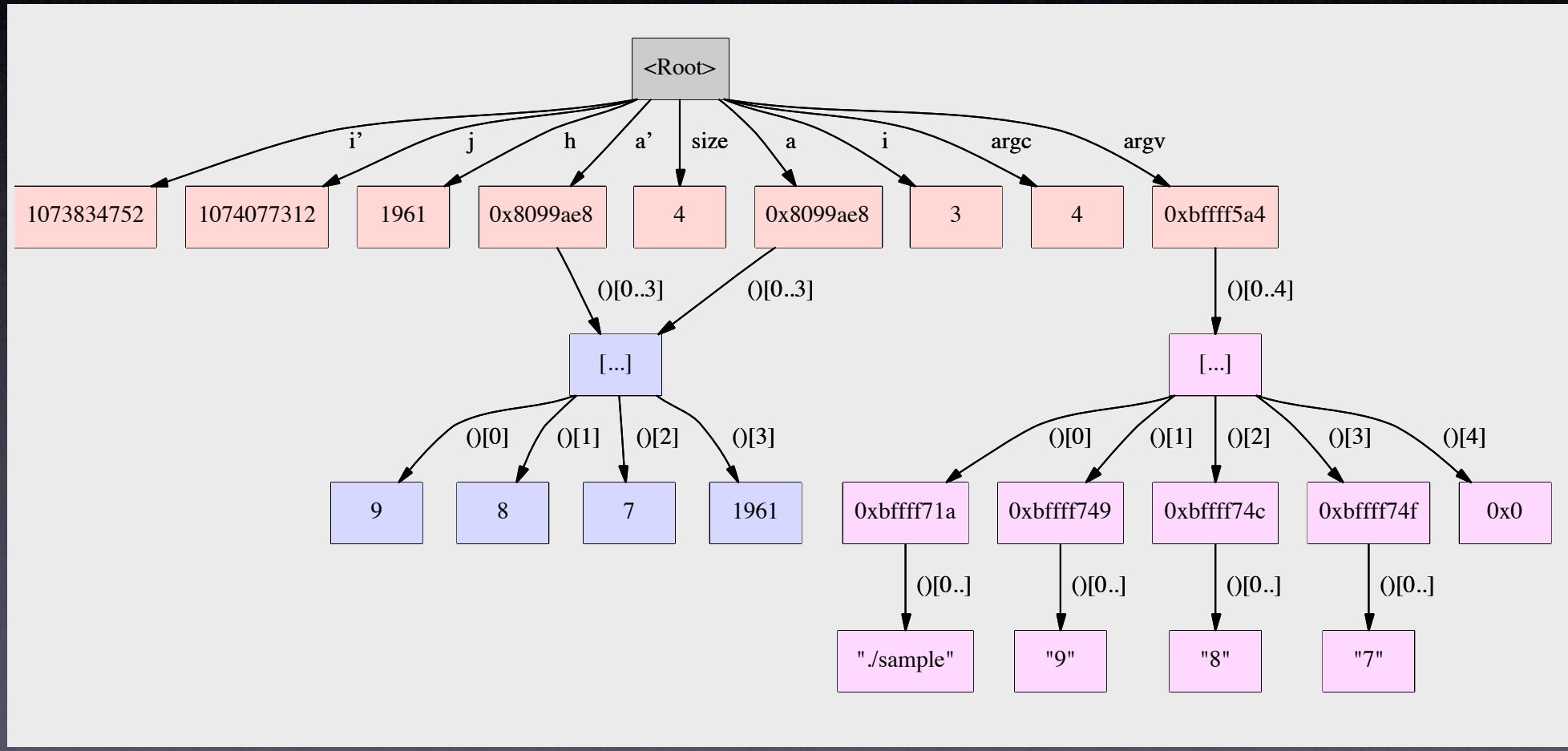
■ = δ is applied, □ = δ is *not* applied

#	$a'[0]$	$a[0]$	$a'[1]$	$a[1]$	$a'[2]$	$a[2]$	*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

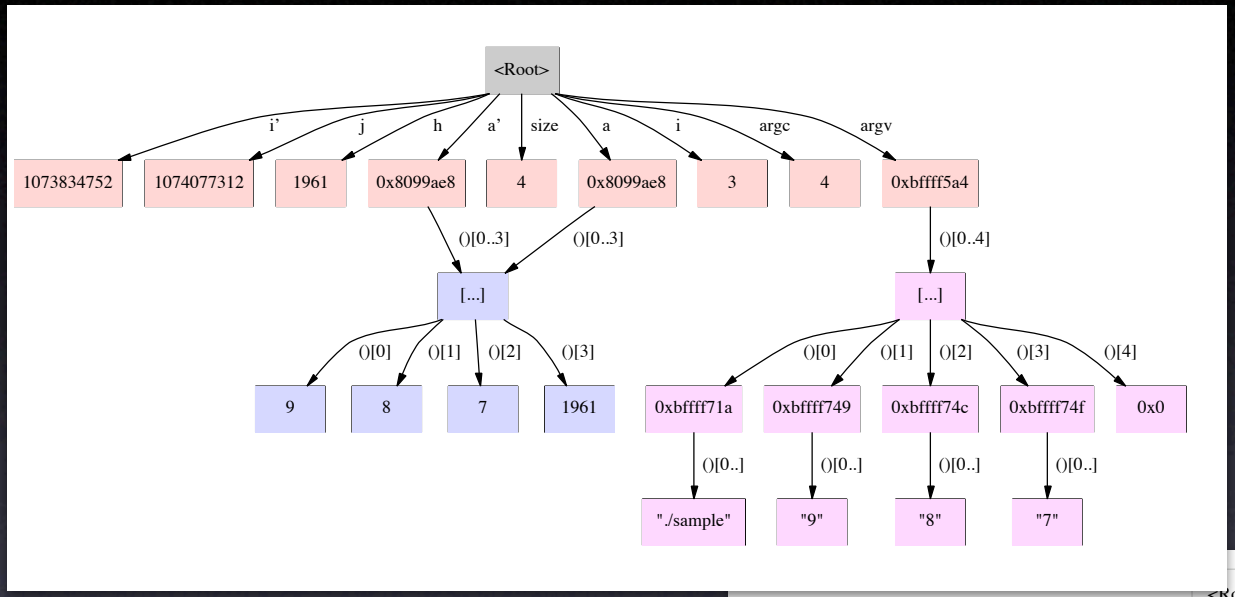
A Memory Graph



Unfolding Memory

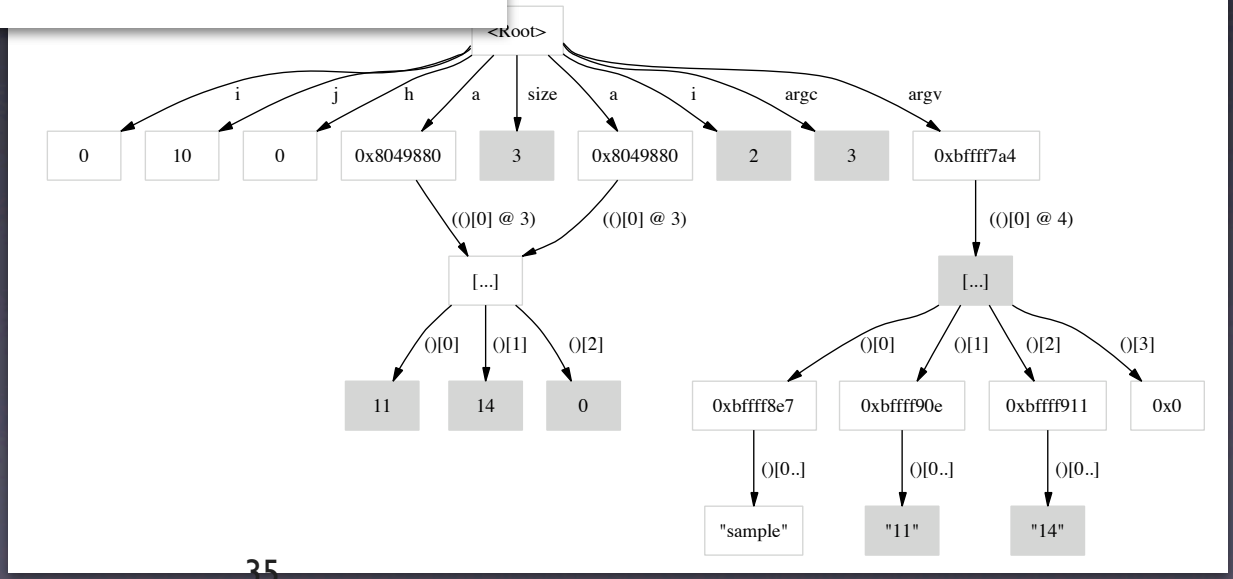
- Any variable: make new node
- Structures: unfold all members
- Arrays: unfold all elements
- Pointers: unfold object being pointed to
  - *Does p point to something? And how many?*

Comparing States



passing run

failing run

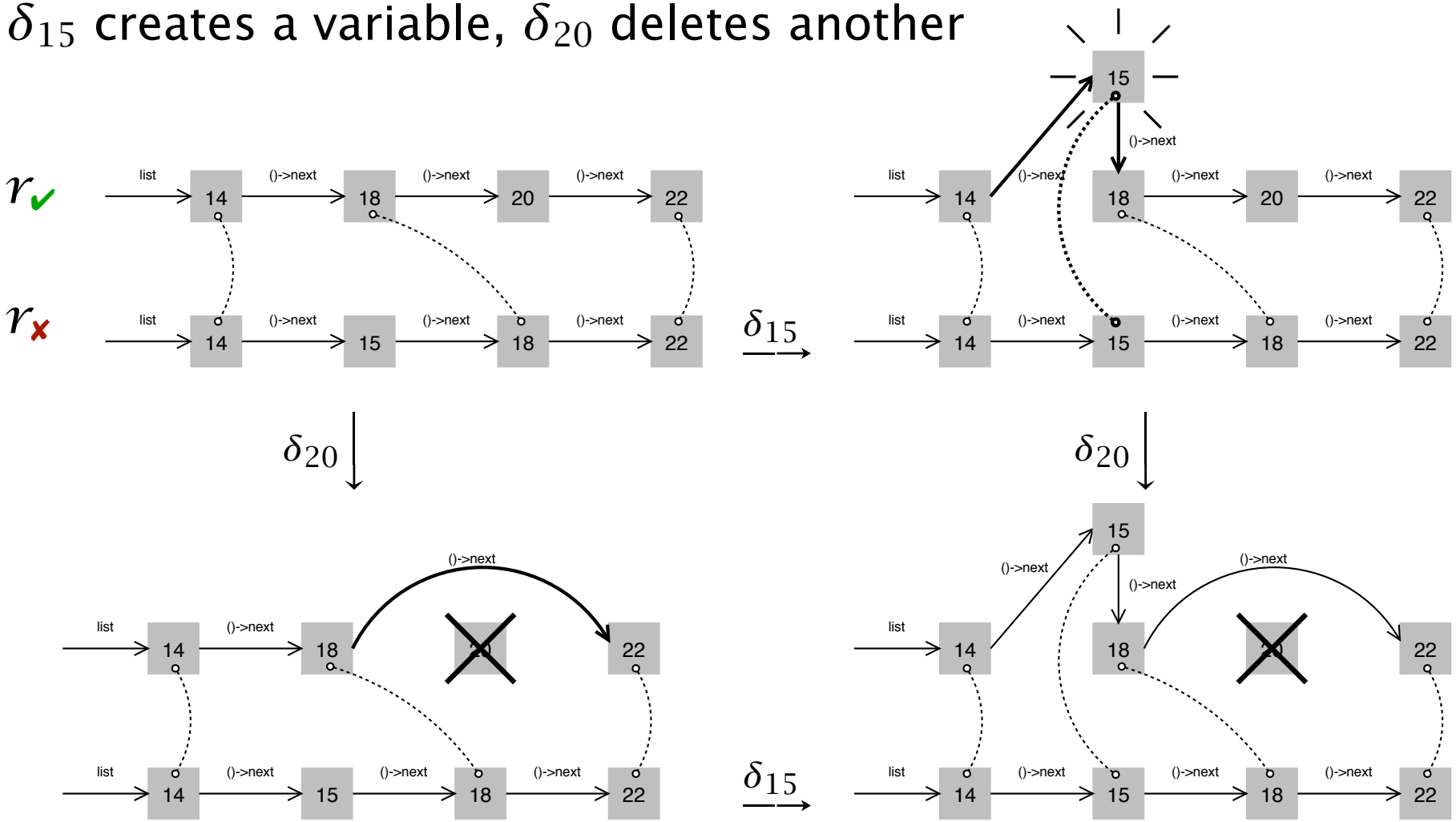


Comparing States

- 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



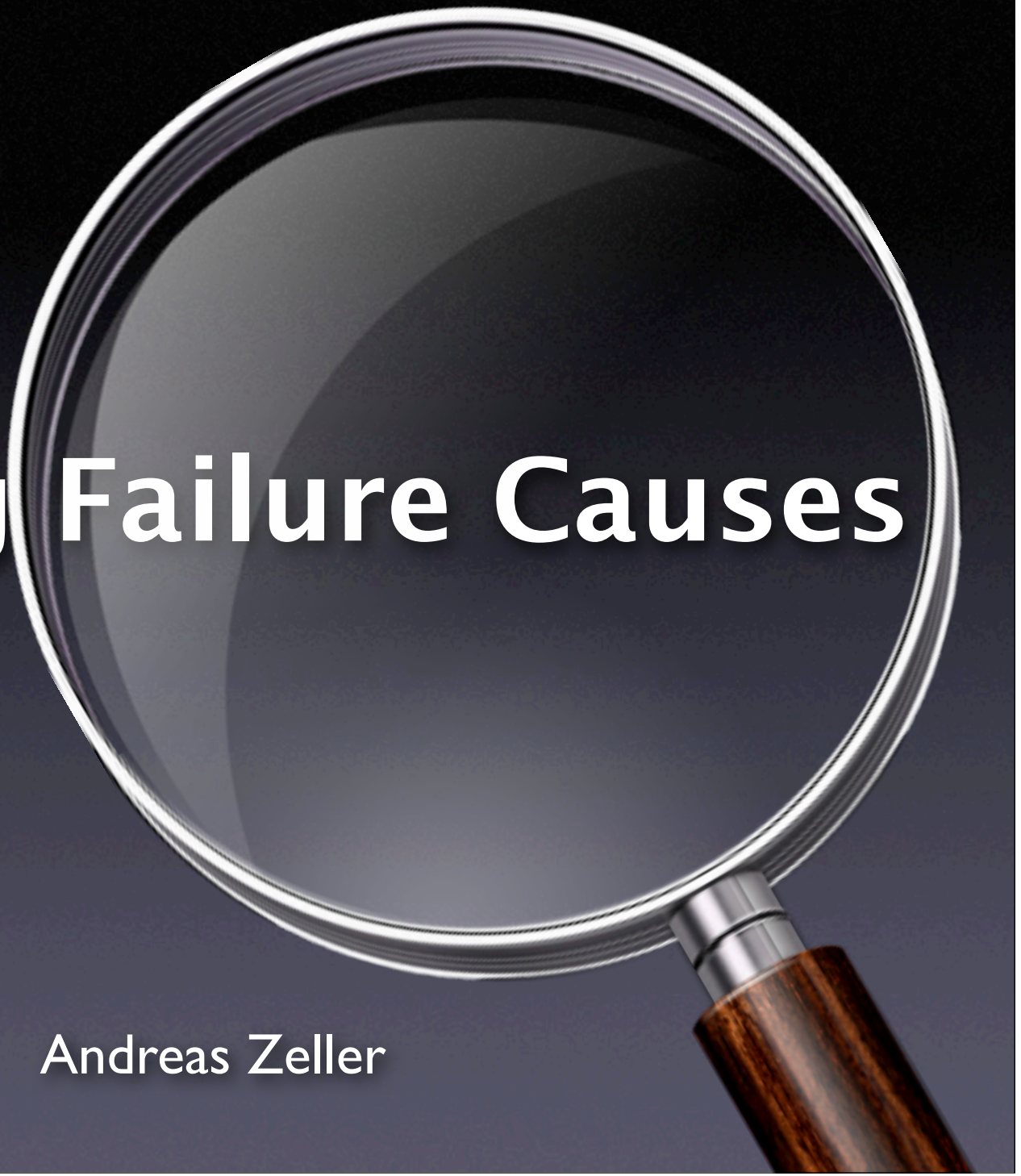
Concepts

- ★ 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.

Locating Failure Causes

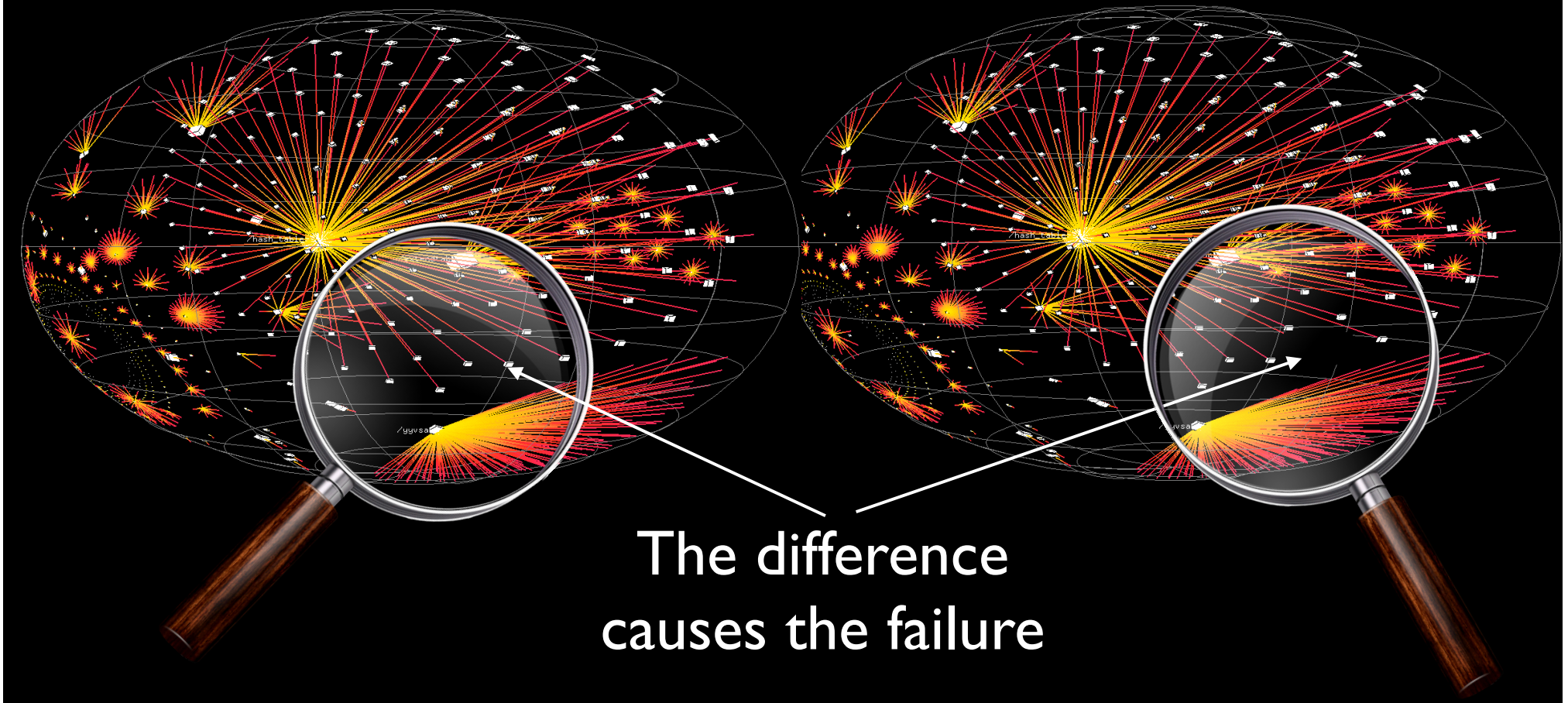


Andreas Zeller

Finding Causes

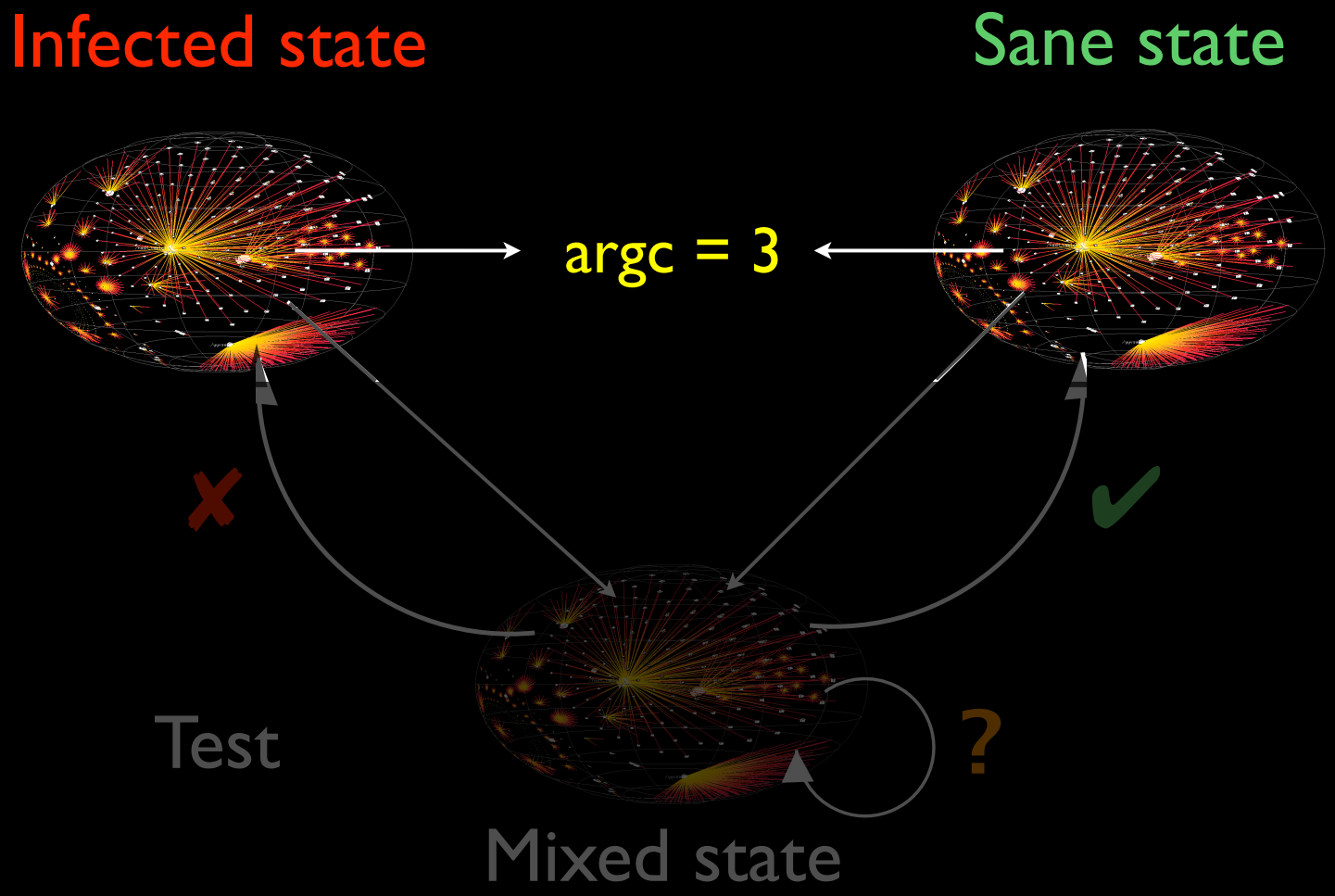
Infected state

Sane state

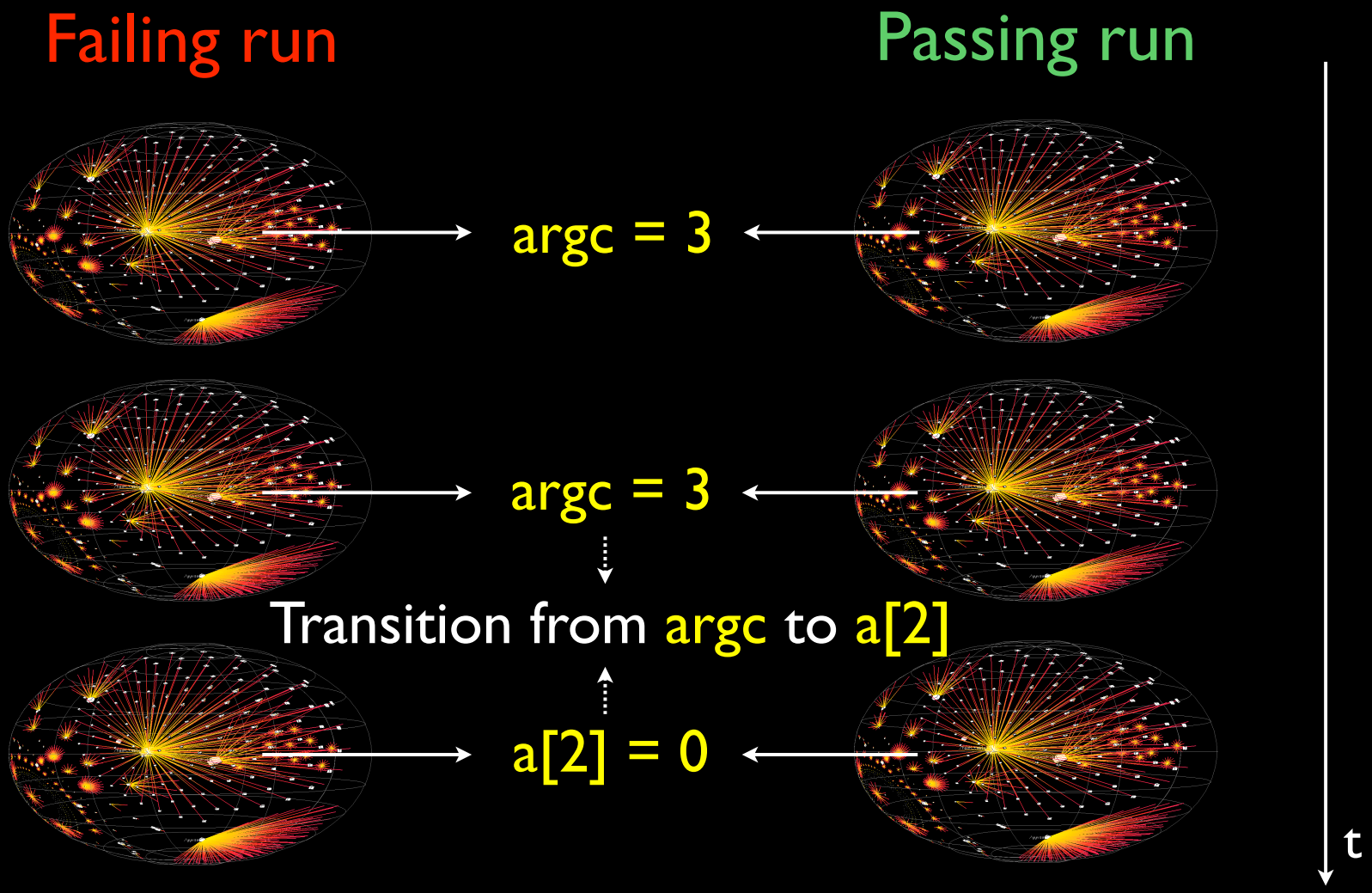


The difference
causes the failure

Search in Space



Search in Time



Transitions

A *cause transition* occurs when a *new variable* begins to be a failure cause:

- argc no longer causes the failure…
- …but a[2] does!

Can be narrowed down by binary search

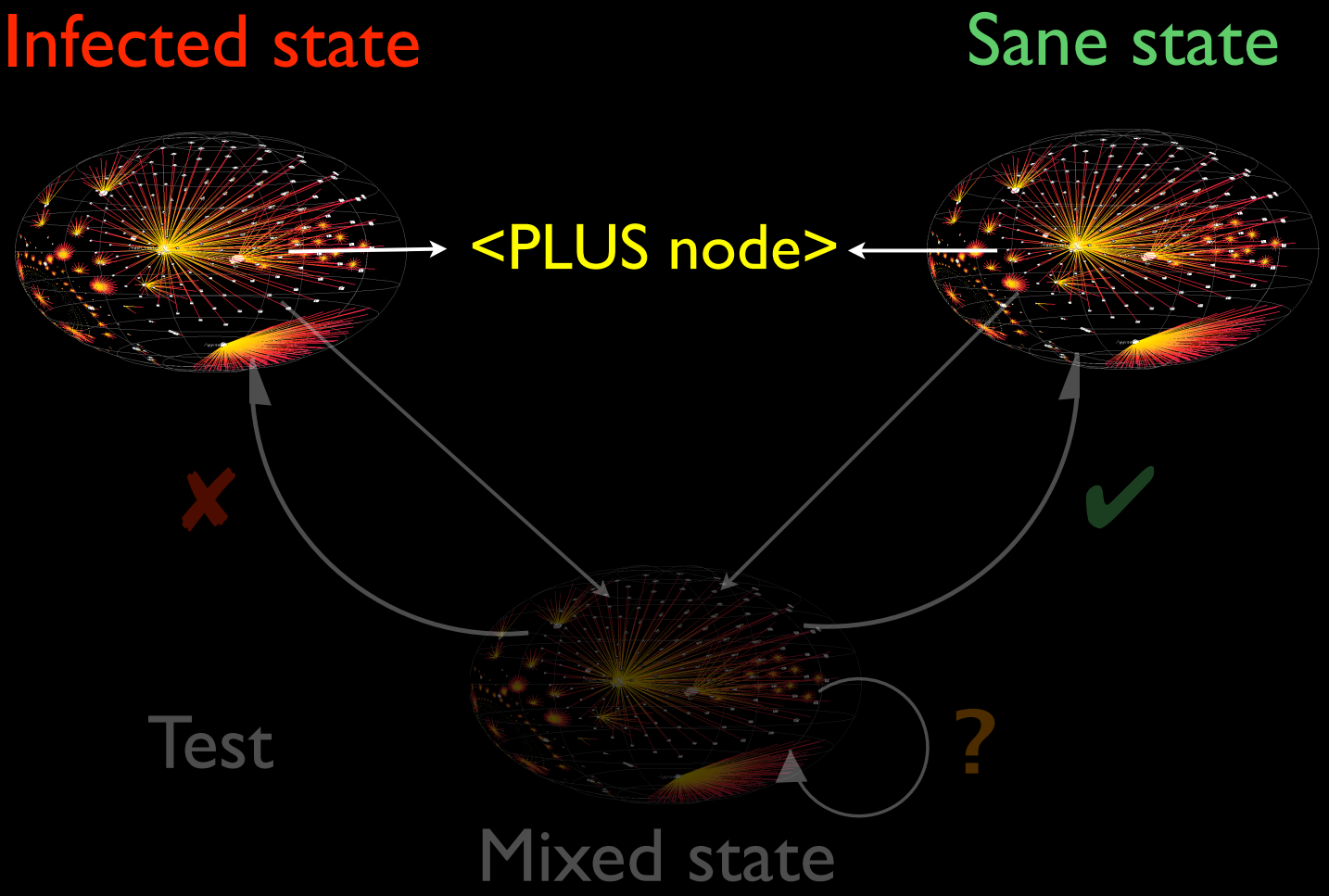
Why Transitions?

- Each failure cause in the program state is caused by some statement
- These statements are executed at cause transitions
- Cause transitions thus are statements that cause the failure!

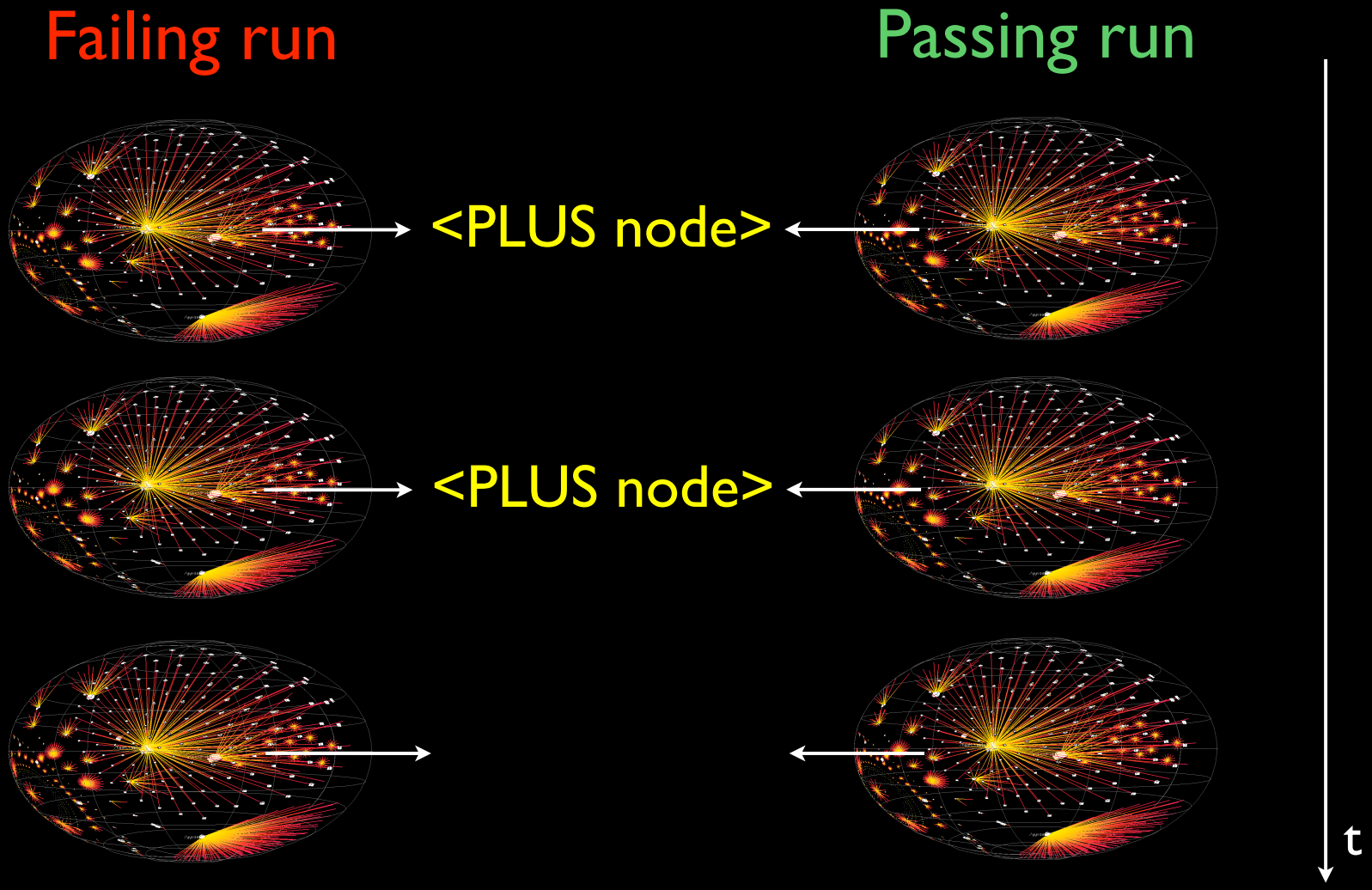
Potential Fixes

- Each cause transition implies a *fix* to make the failure no longer occur – just prohibit the transition
- A cause transition is more than a potential fix – it may be "the" defect itself

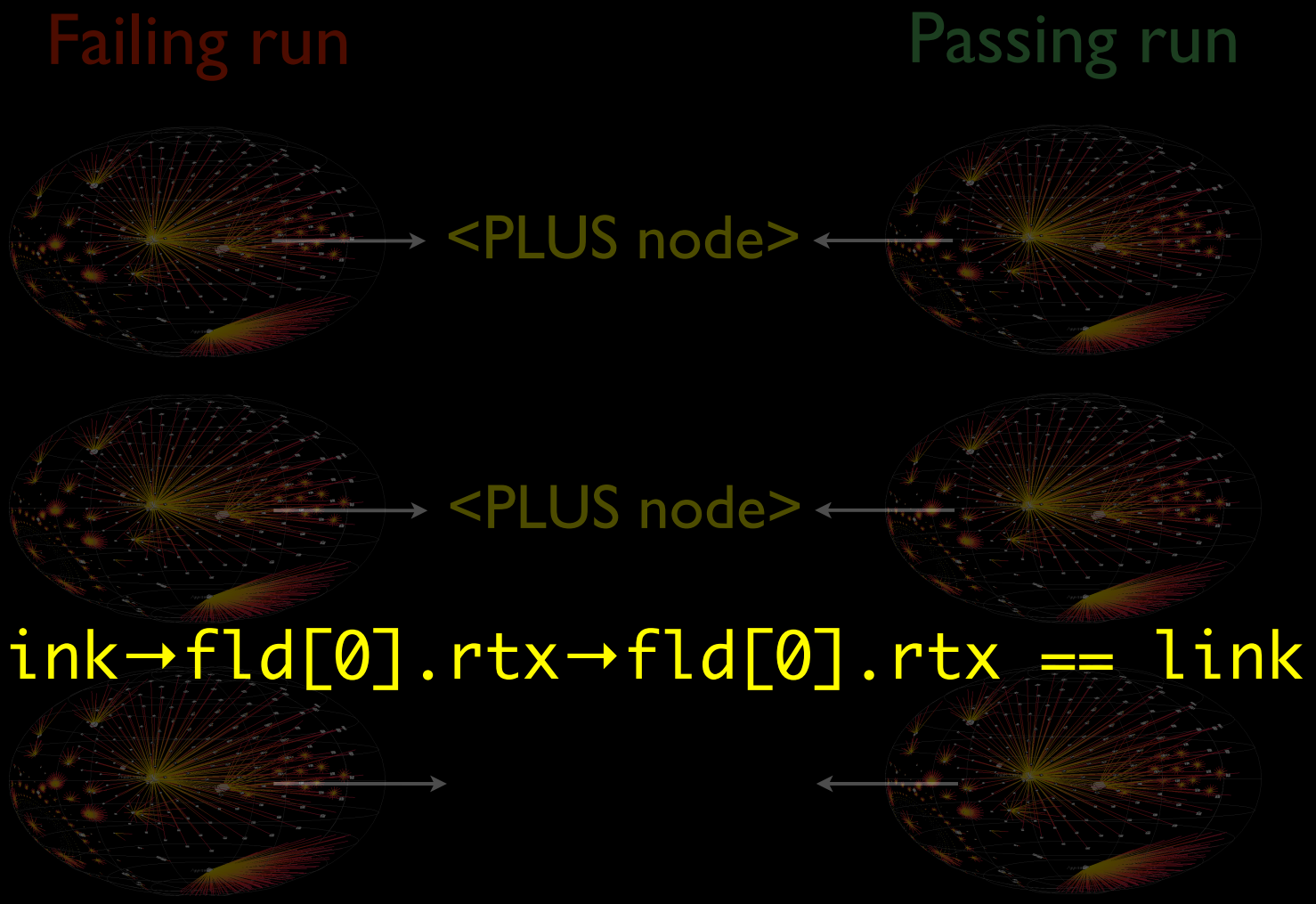
Searching GCC State



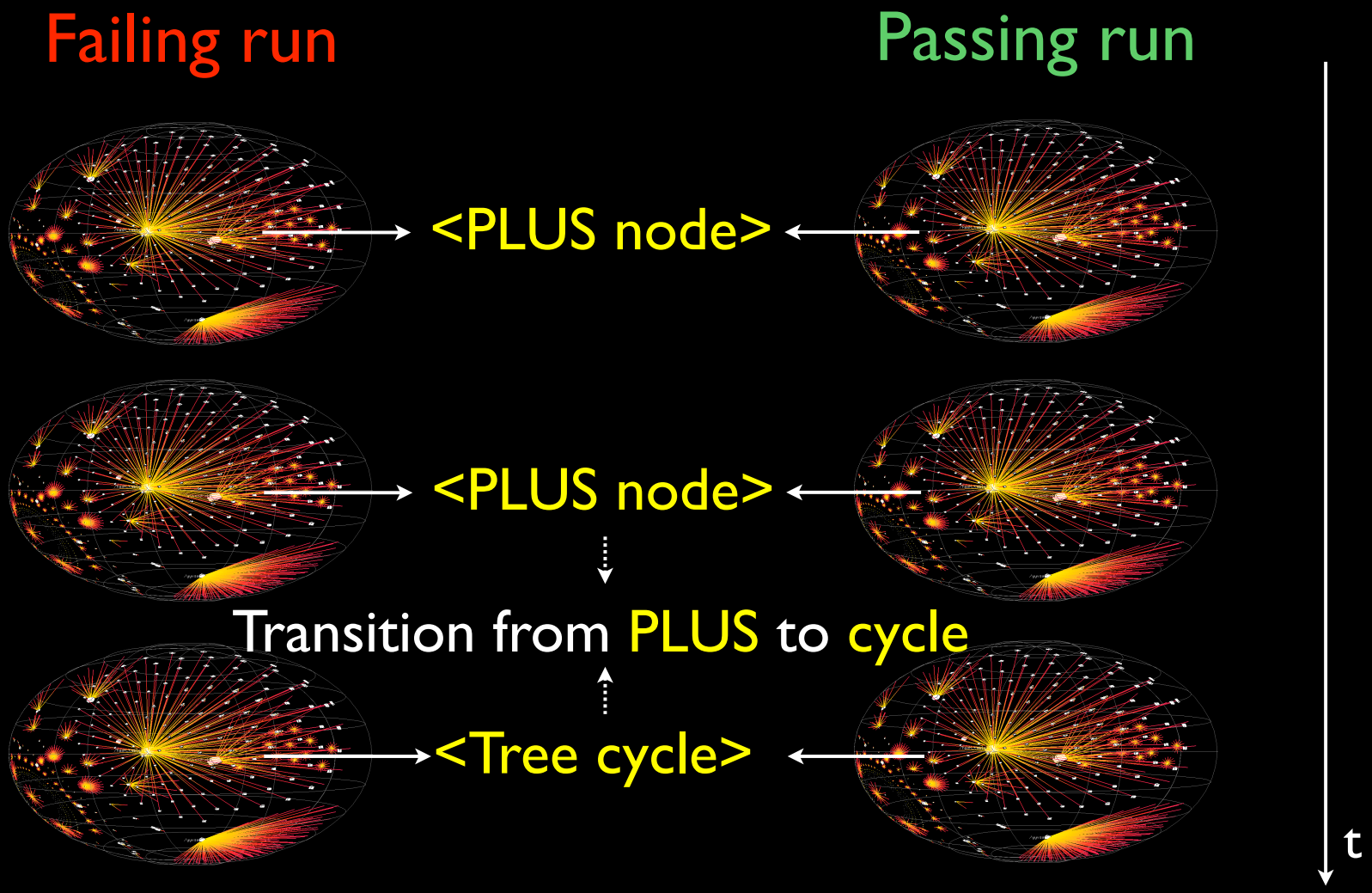
Search in Time



Search in Time



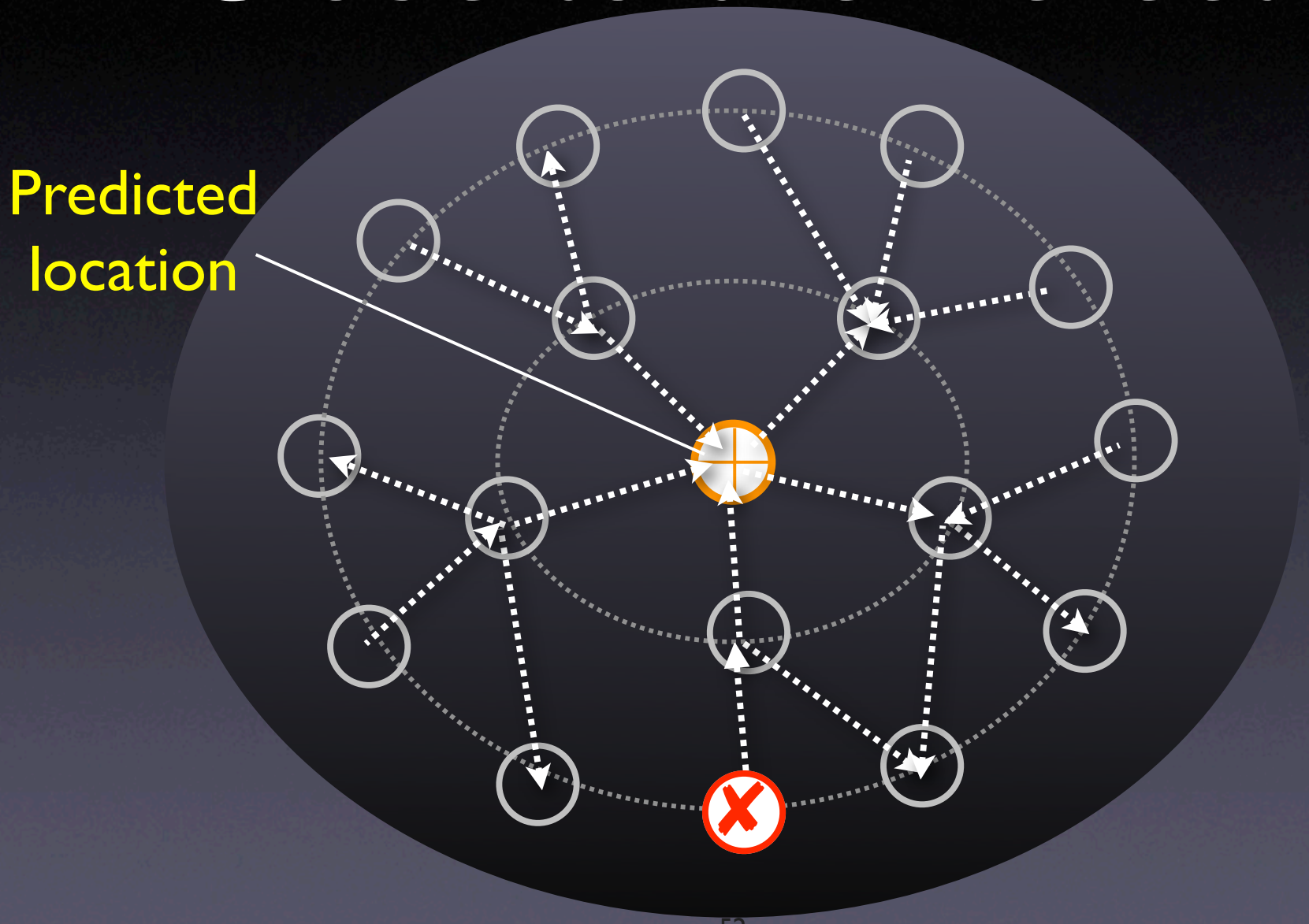
Search in Time



All 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`

Close to the Defect



Concepts (3)

- Cause transition statements can be identified using a binary search.
- Cause transition statements are potential places where a programmers fix code to prevent a failure.

Preview for This Wednesday

- Understanding Regression Testing Selection, Prioritization, Augmentation, and Minimization.
- Path Spectra (The use of program profiling for software maintenance with applications to Y2K problem)
  - Sidd (advocate)
  - Srinivas (skeptic)

Announcement

- Everyone who came to class and claimed your quiz or past class activities will receive one class participation point today.