Isolating Cause–Effect Chains

Andreas Zeller

bug.c

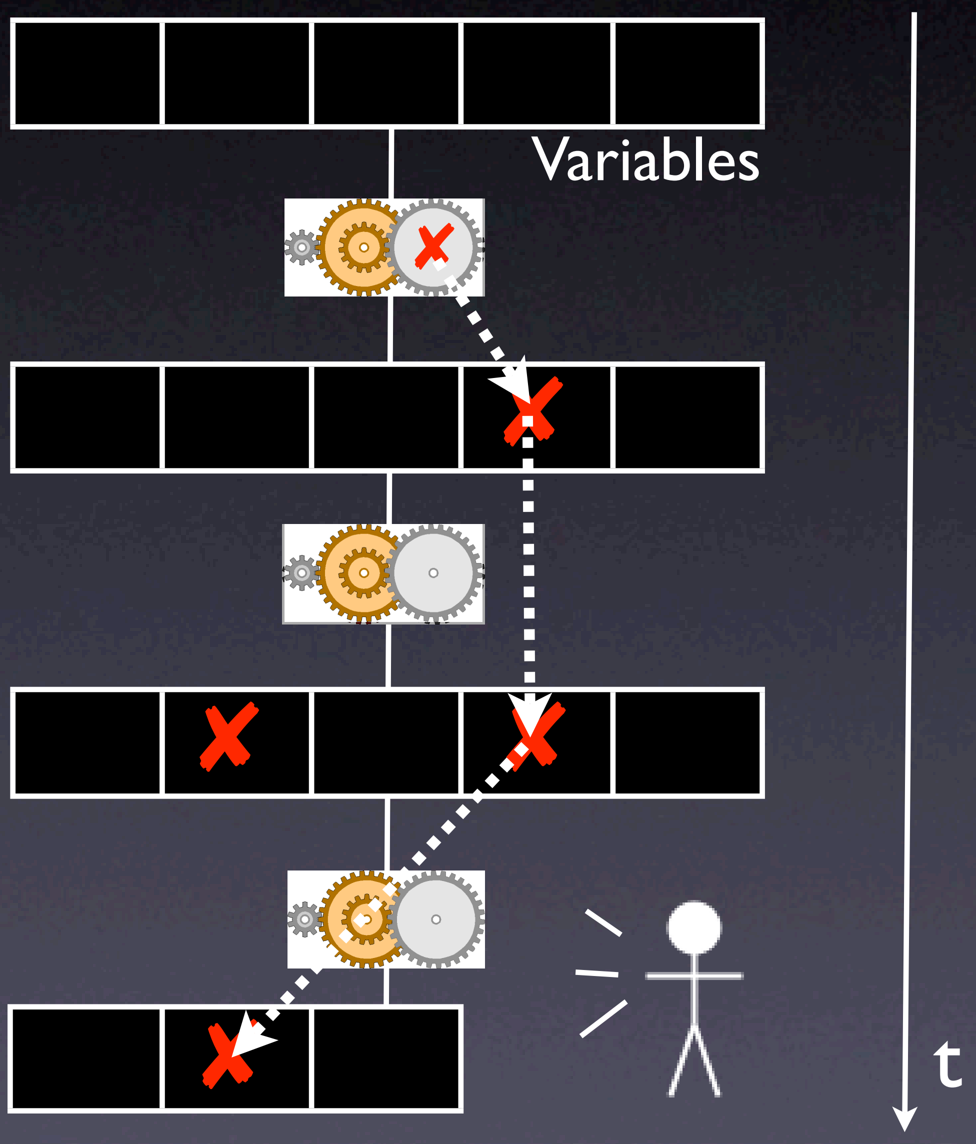
```
double bug(double z[], int n) {
    int i, j;

    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

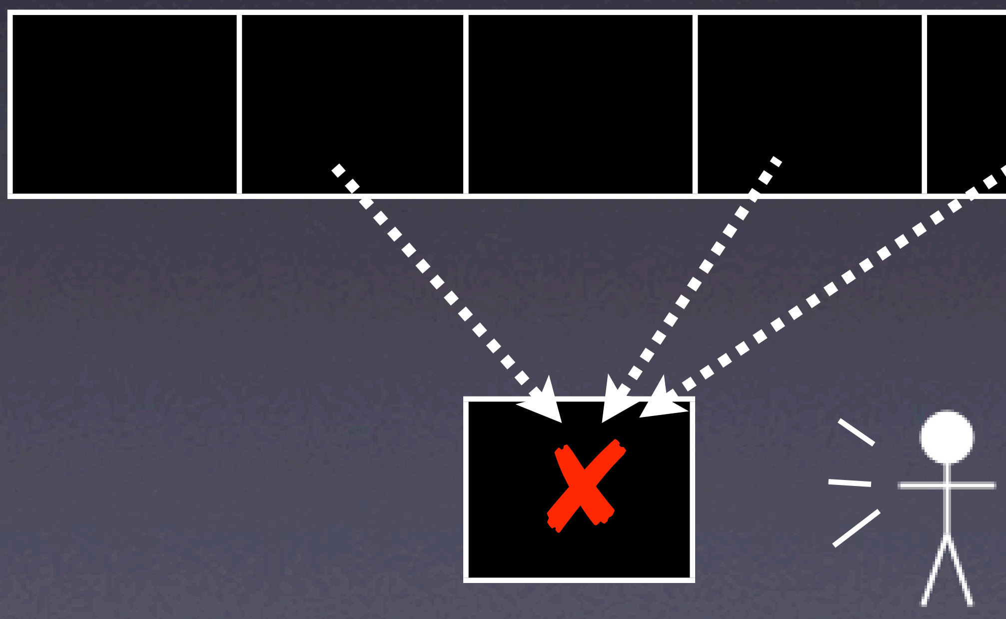
1. The programmer creates a *defect* – an error in the code.
2. When executed, the defect creates an *infection* – an error in the state.
3. The infection *propagates*.
4. The infection causes a *failure*.

This infection chain must be traced back – and broken.

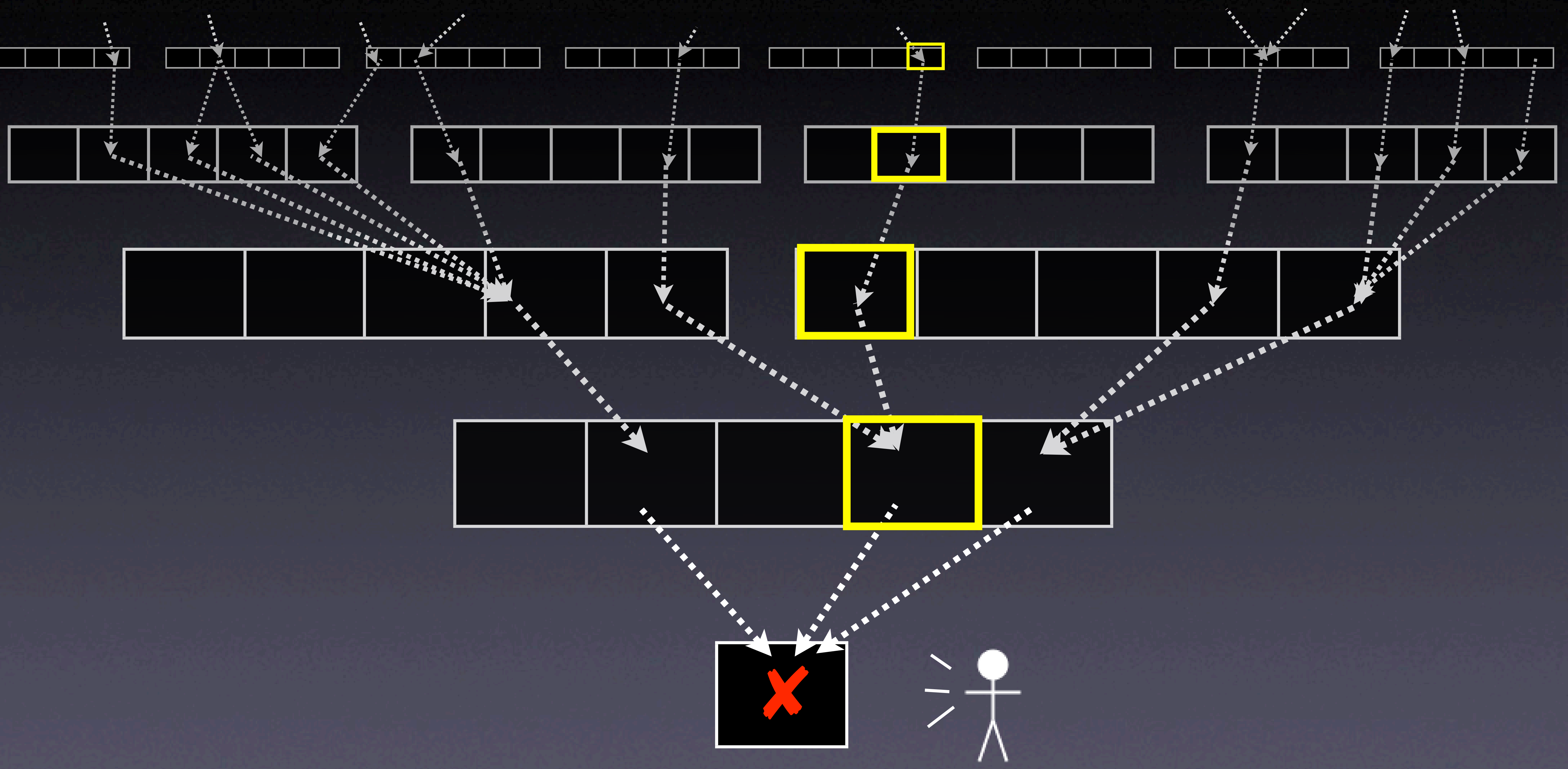


Tracing Infections

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



Tracing Infections



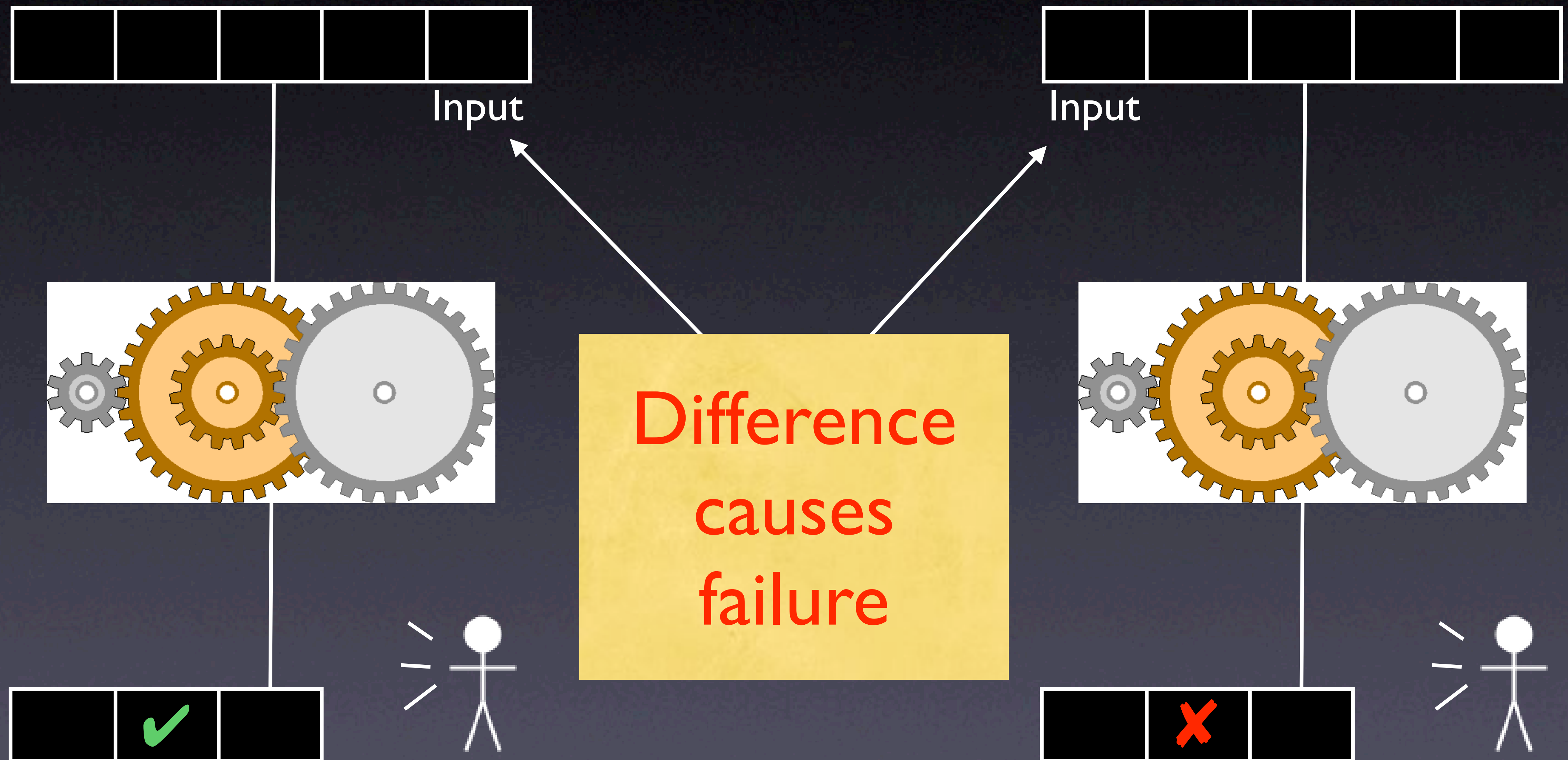
Real Code

- Opaque – e.g. third-party code
- Parallel – threads and processes
- Distributed – across multiple machines
- Dynamic – e.g. reflection in Java
- Multilingual – say, Python + C + SQL

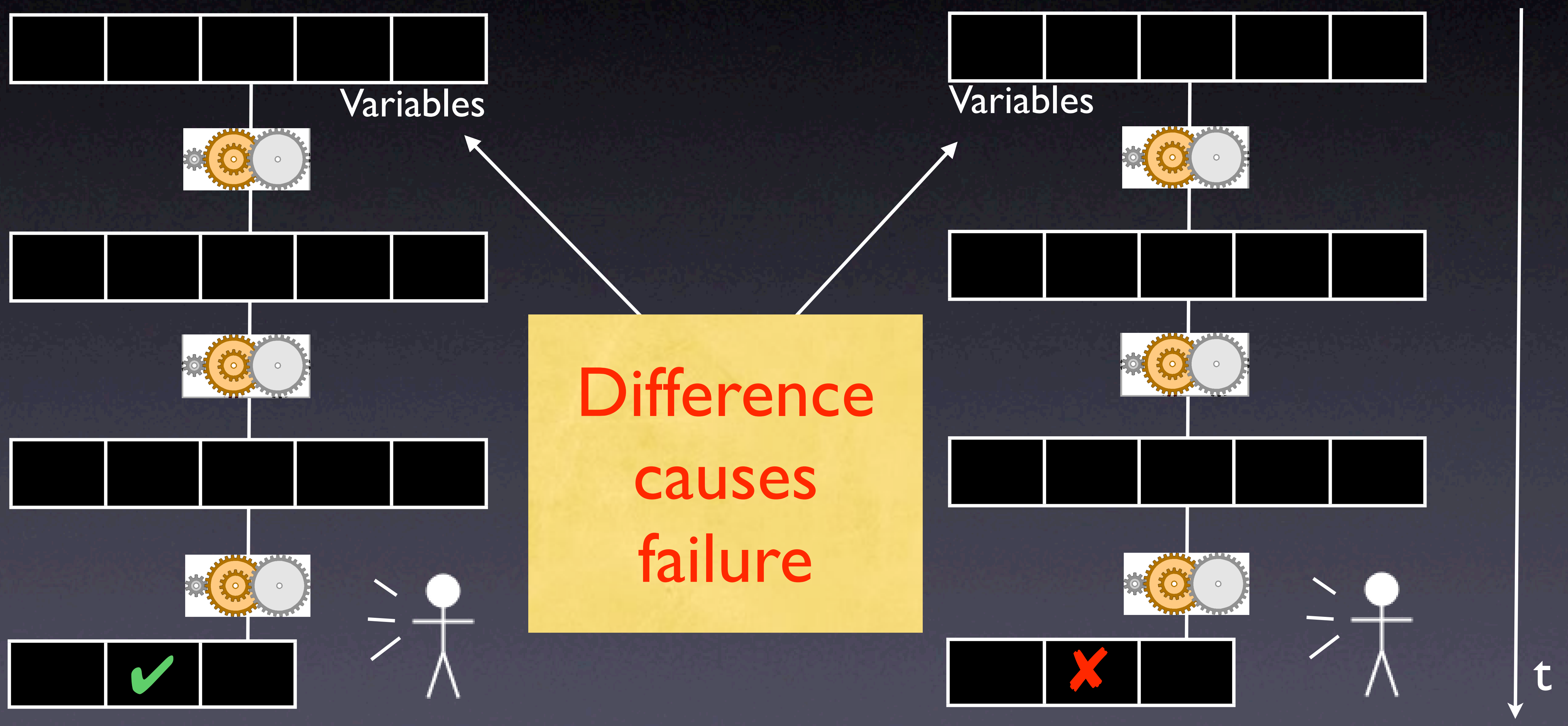
Obscure Code

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

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

Demo

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

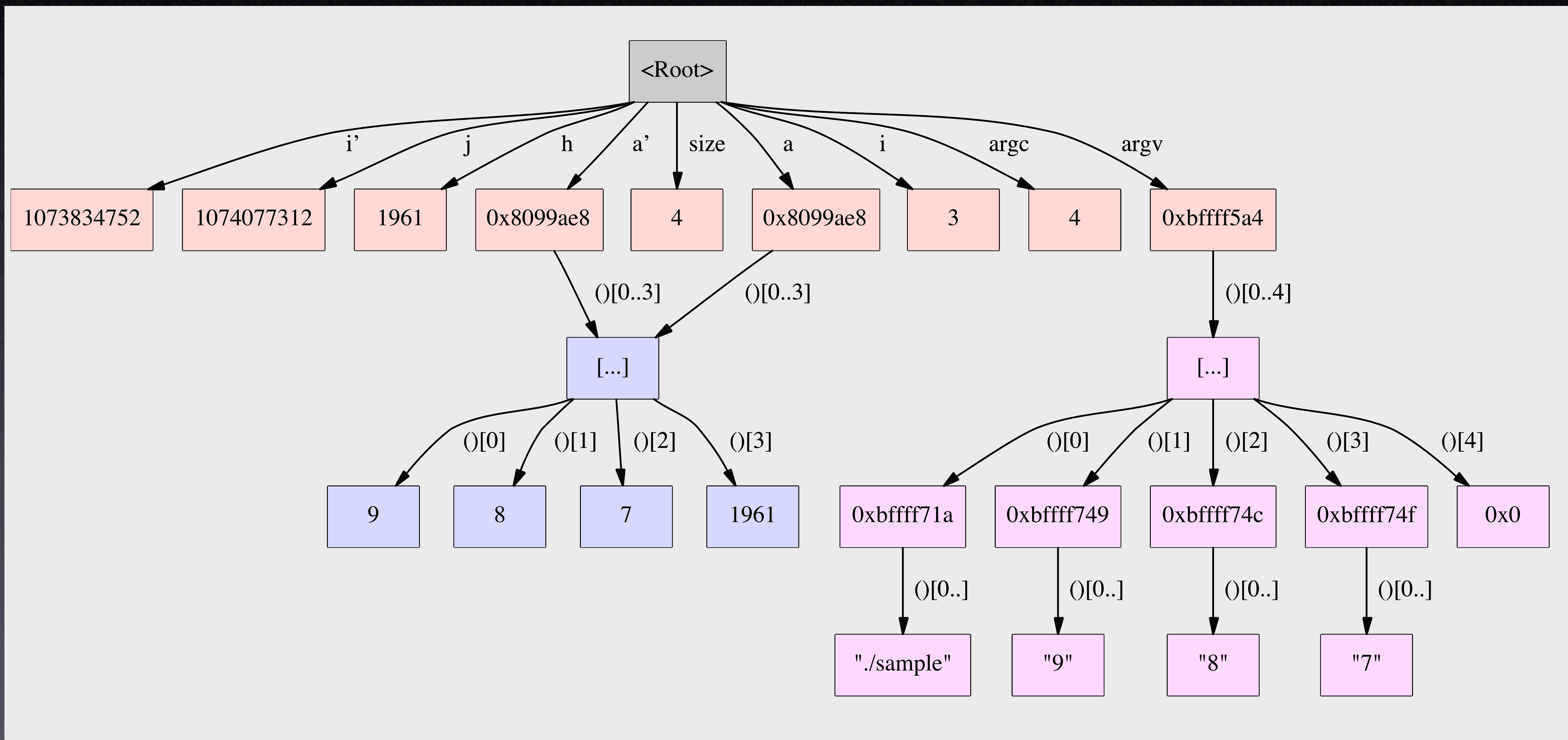
■ = δ is applied, □ = δ is *not* applied

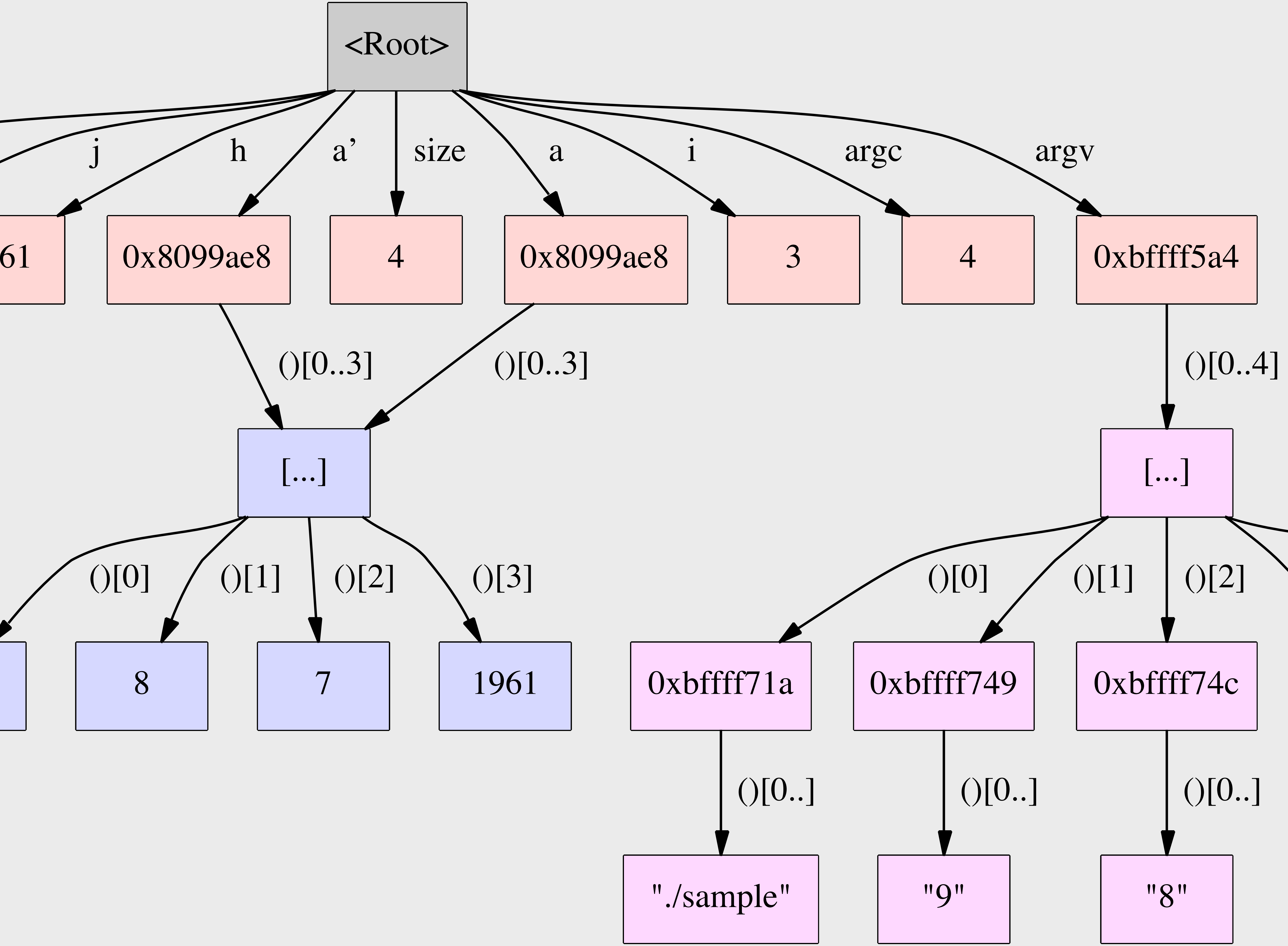
#	$a'[0]$	$a[0]$	$a'[1]$	$a[1]$	$a'[2]$	$a[2]$	$argc$	$argv[1]$	$argv[2]$	$argv[3]$	i	$size$	Output	Test
1	□	□	□	□	□	□	□	□	□	□	□	□	7 8 9	✔
2	■	■	■	■	■	■	■	■	■	■	■	■	0 11	✘
3	■	■	■	■	■	■	□	□	□	□	□	□	0 11 14	✘
4	■	■	■	□	□	□	□	□	□	□	□	□	7 11 14	?
5	□	□	□	■	■	■	□	□	□	□	□	□	0 9 14	✘
6	□	□	□	■	□	□	□	□	□	□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

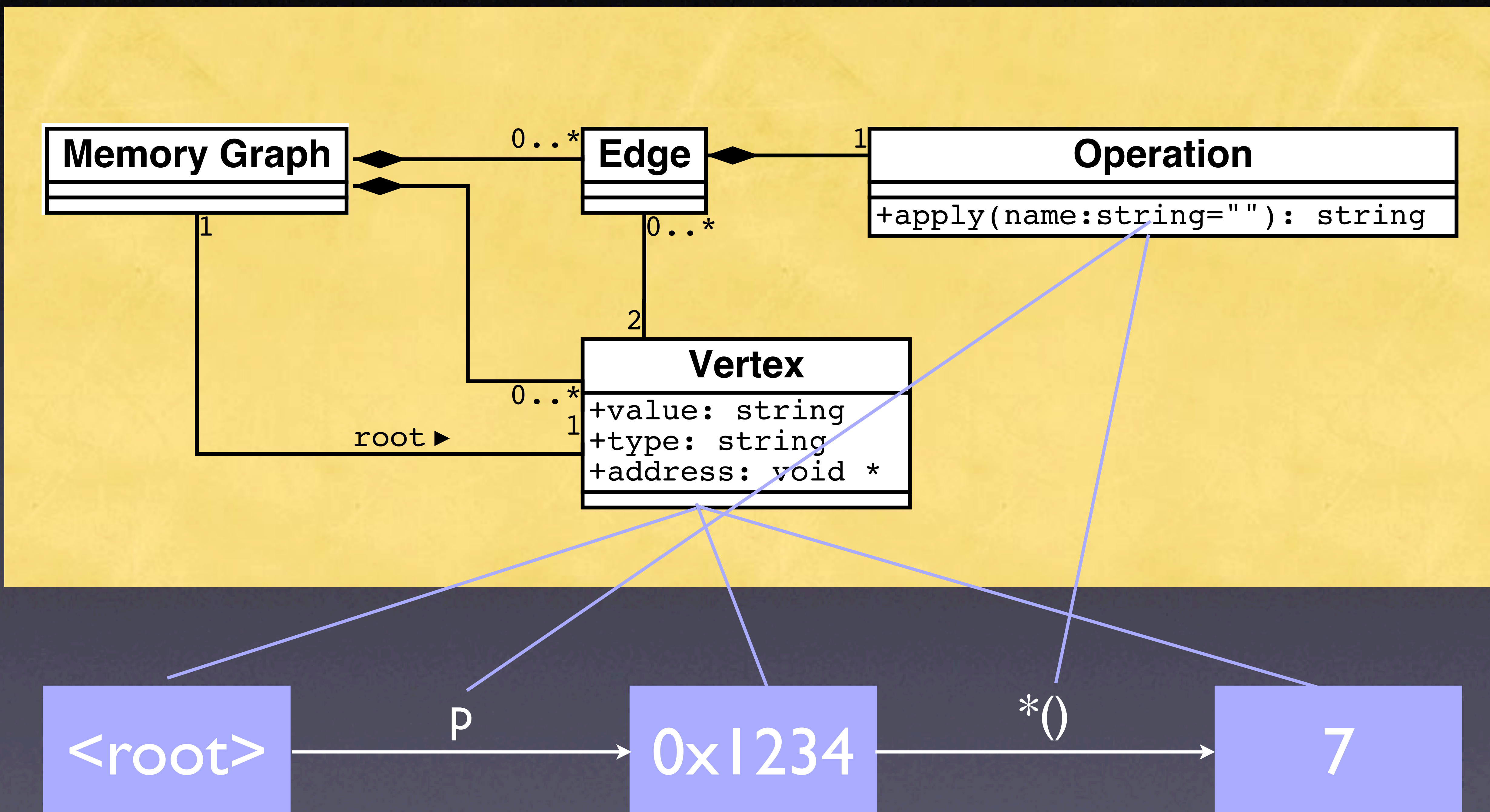
A Memory Graph





<Root>
j
h
a'
size
a
i
argc
argv
61
0x8099ae8
4
0x8099ae8
3
4
0xbffff5a4
()[0..3]
()[0..3]
()[0..4]
[...]
[...]
()[0]
()[1]
()[2]
()[3]
()[0]
()[1]
()[2]
8
7
1961
0xbffff71a
0xbffff749
0xbffff74c
()[0..]
()[0..]
()[0..]
"./sample"
"9"
"8"

Structure


Memory Graph
Edge
Operation
+apply(name:string=""): string
0..*
1
0..*
2
Vertex
+value: string
+type: string
+address: void *
0..*
1
1
root ▶
<root>
P
0x1234
*()
7

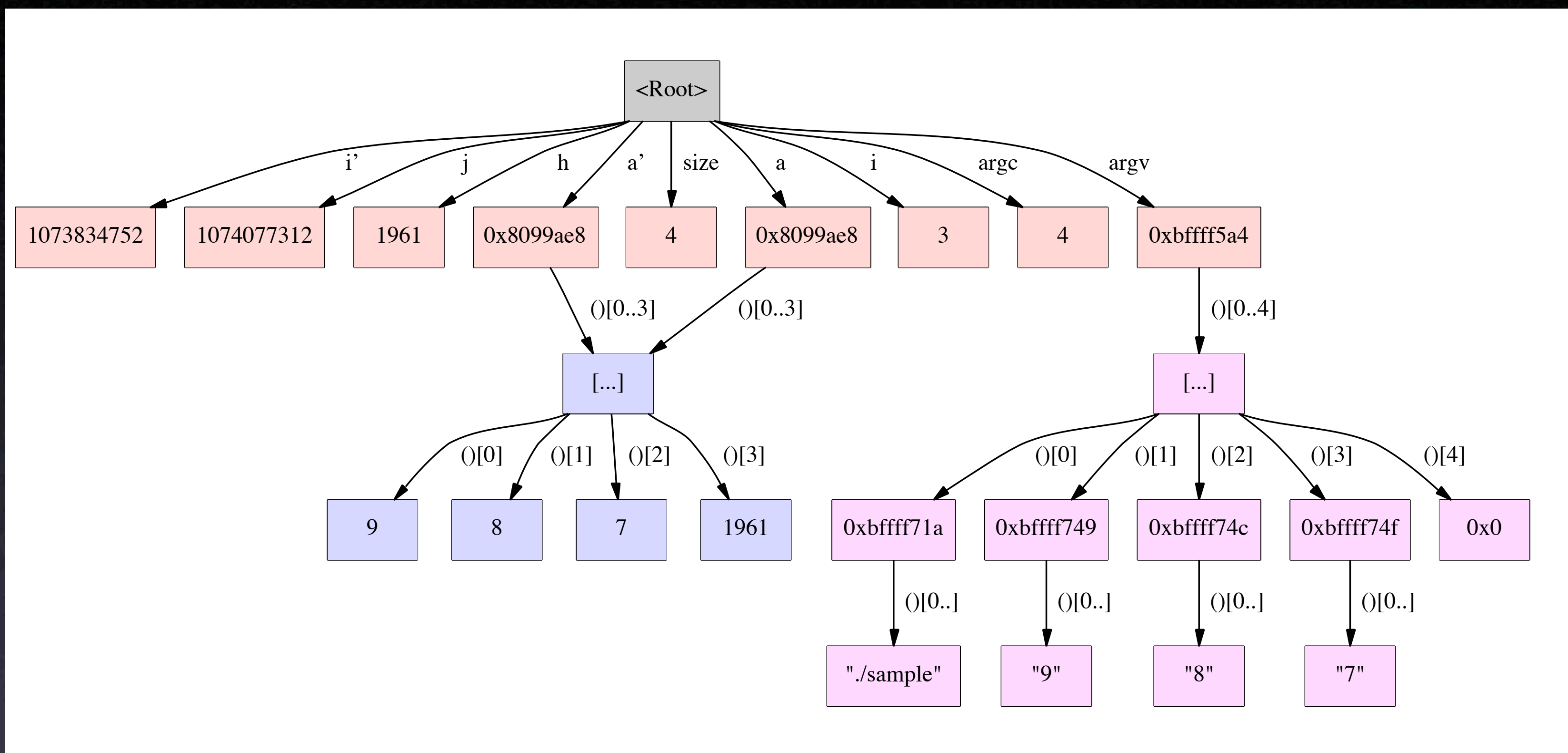
Construction

- Start with <root> node and base variables
  - *Base variables are on the stack and at fixed locations*
- Expand all references, checking for aliases…
- …until all accessible variables are unfolded

Unfolding Memory

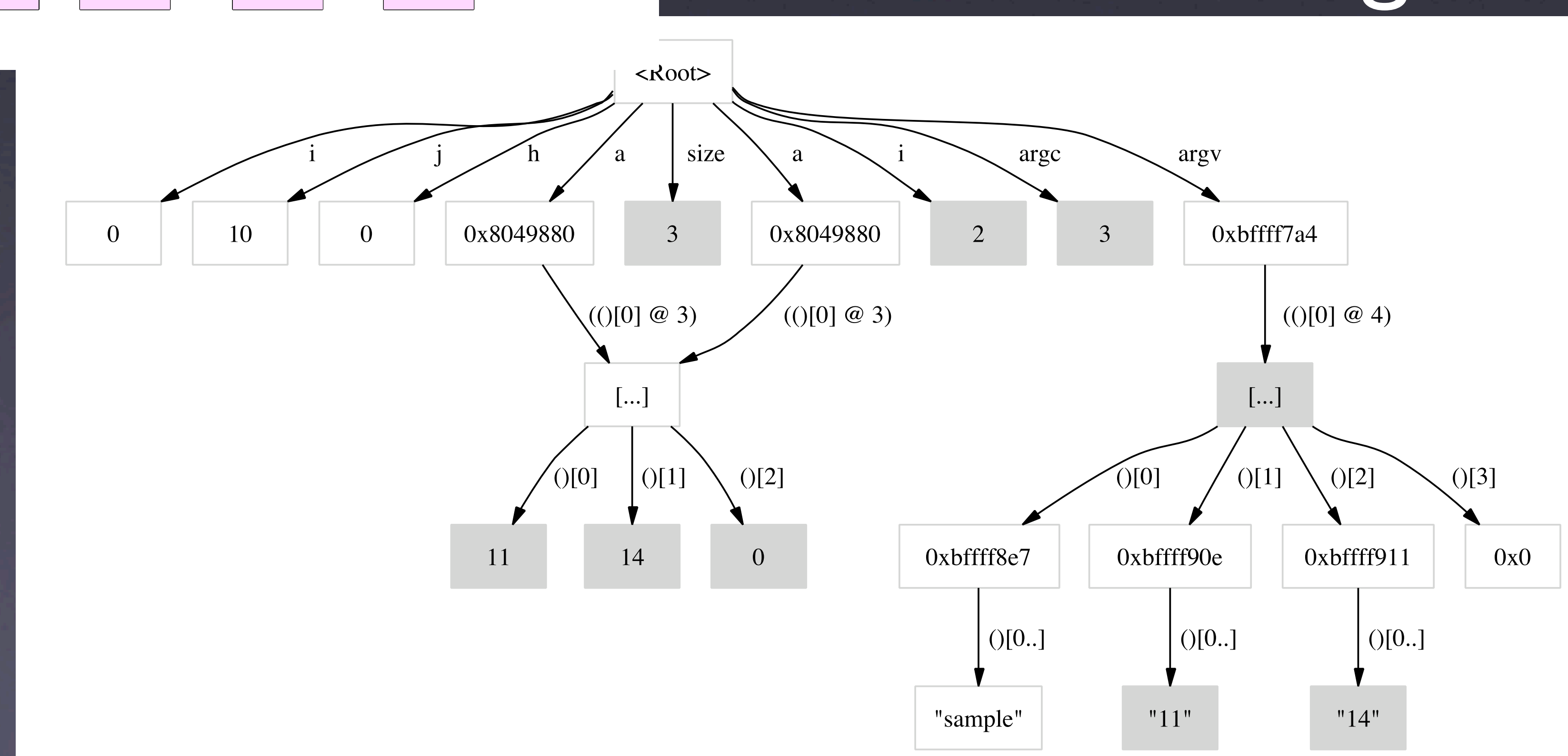
- Any variable: make new node
- Structures: unfold all members
- Arrays: unfold all elements
- Pointers: unfold object being pointed to
  - *Does p point to something? And how many?*

Comparing States



passing run

failing run

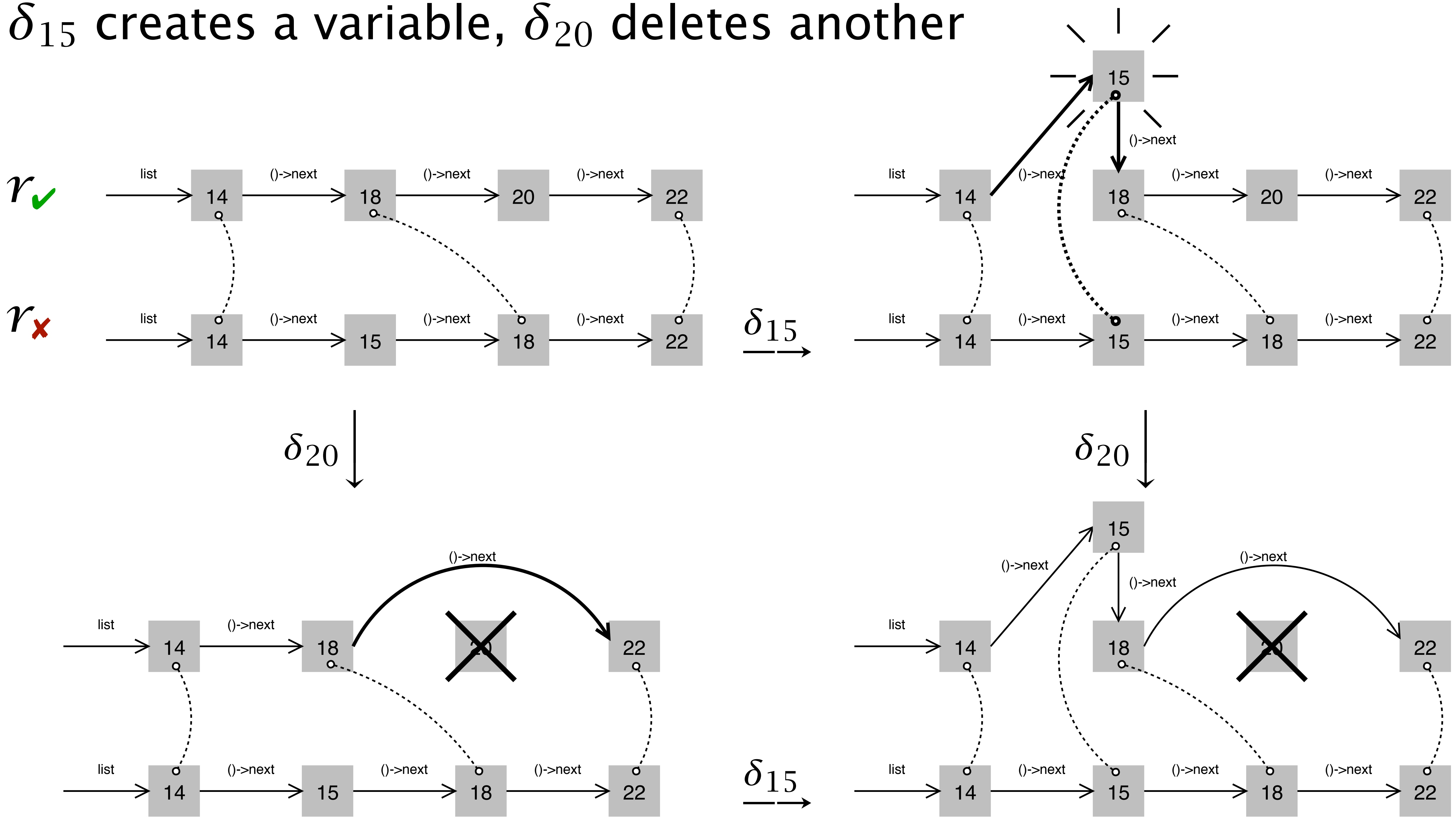


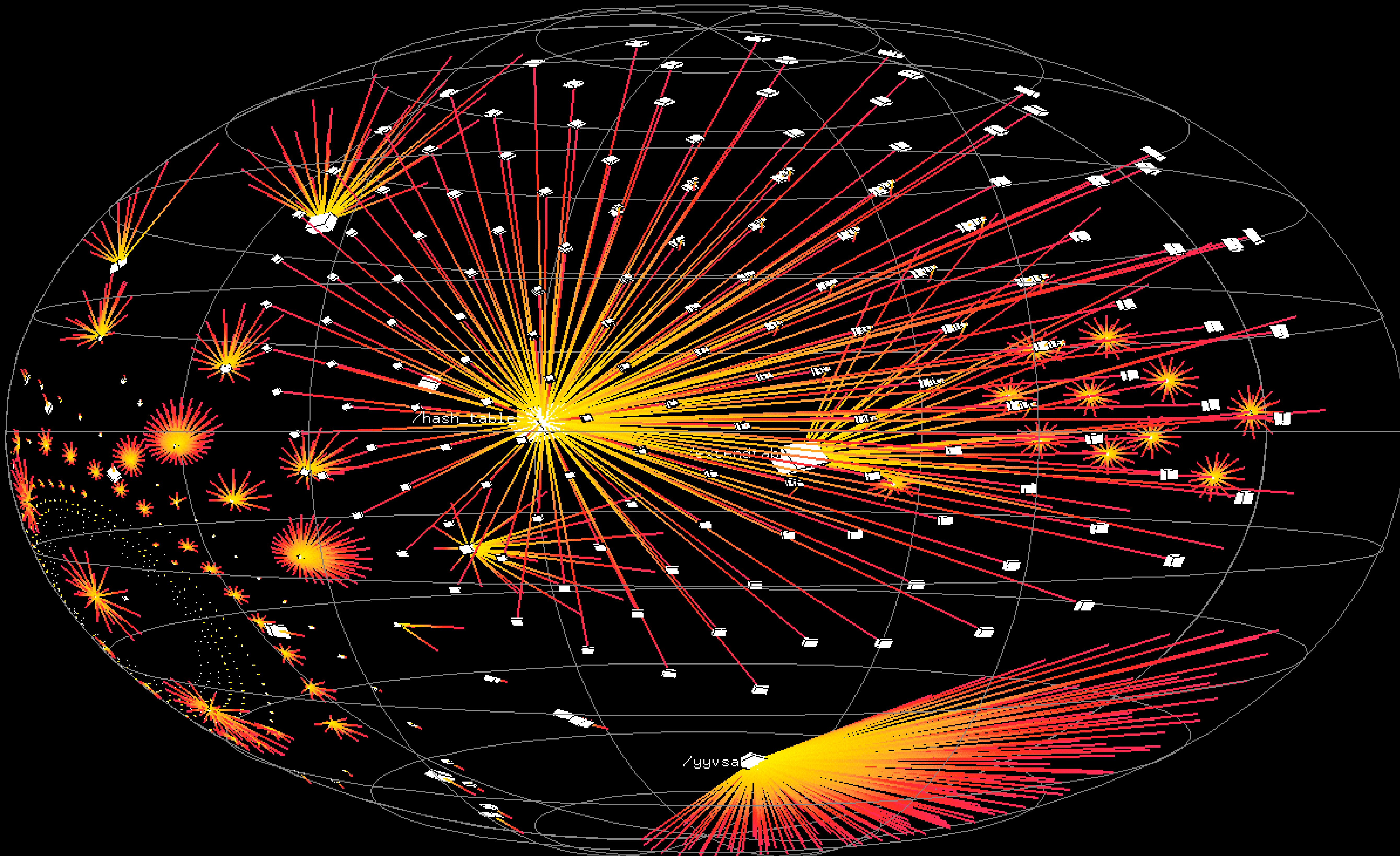
Comparing States

- 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

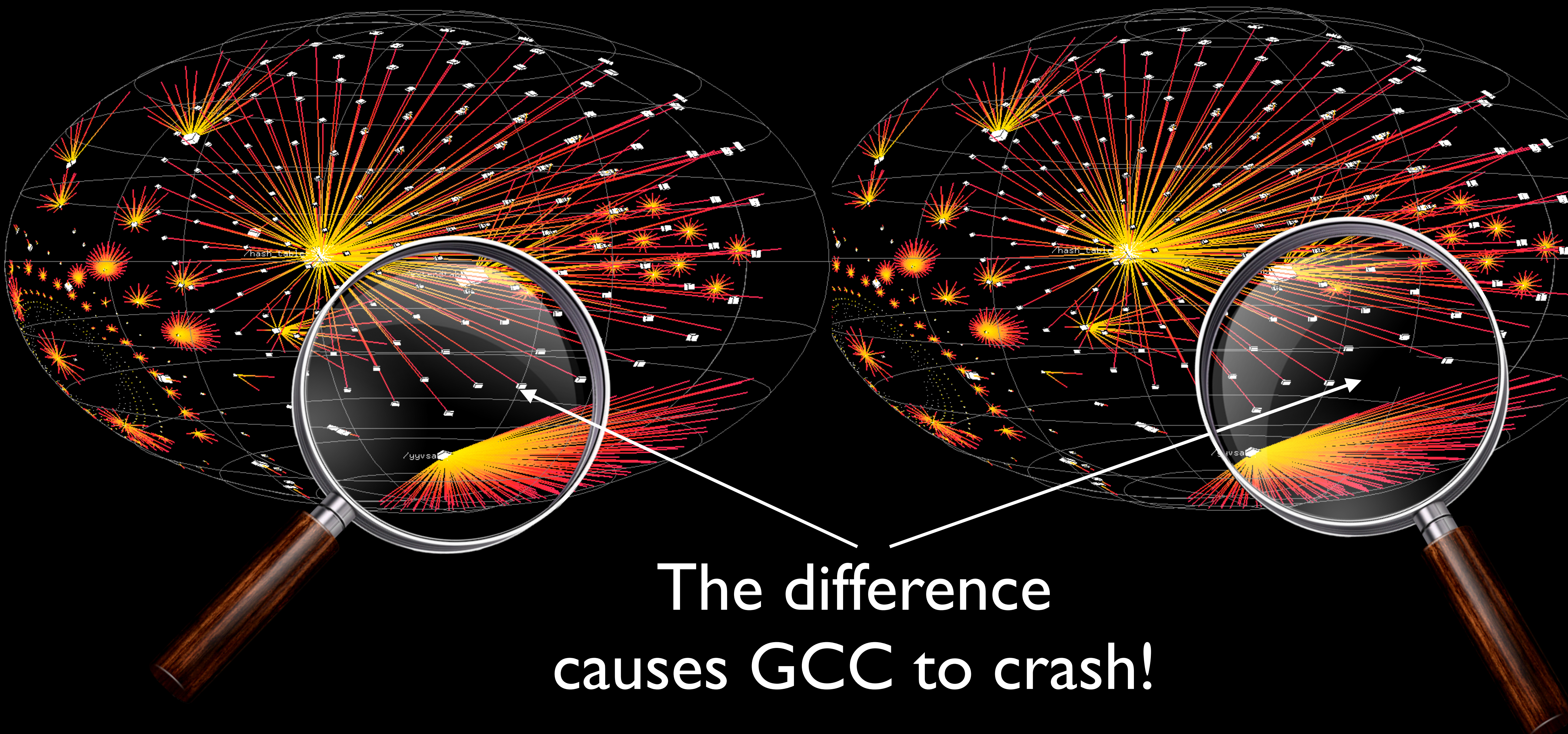



/hash_table
/yyvsa

Causes in State

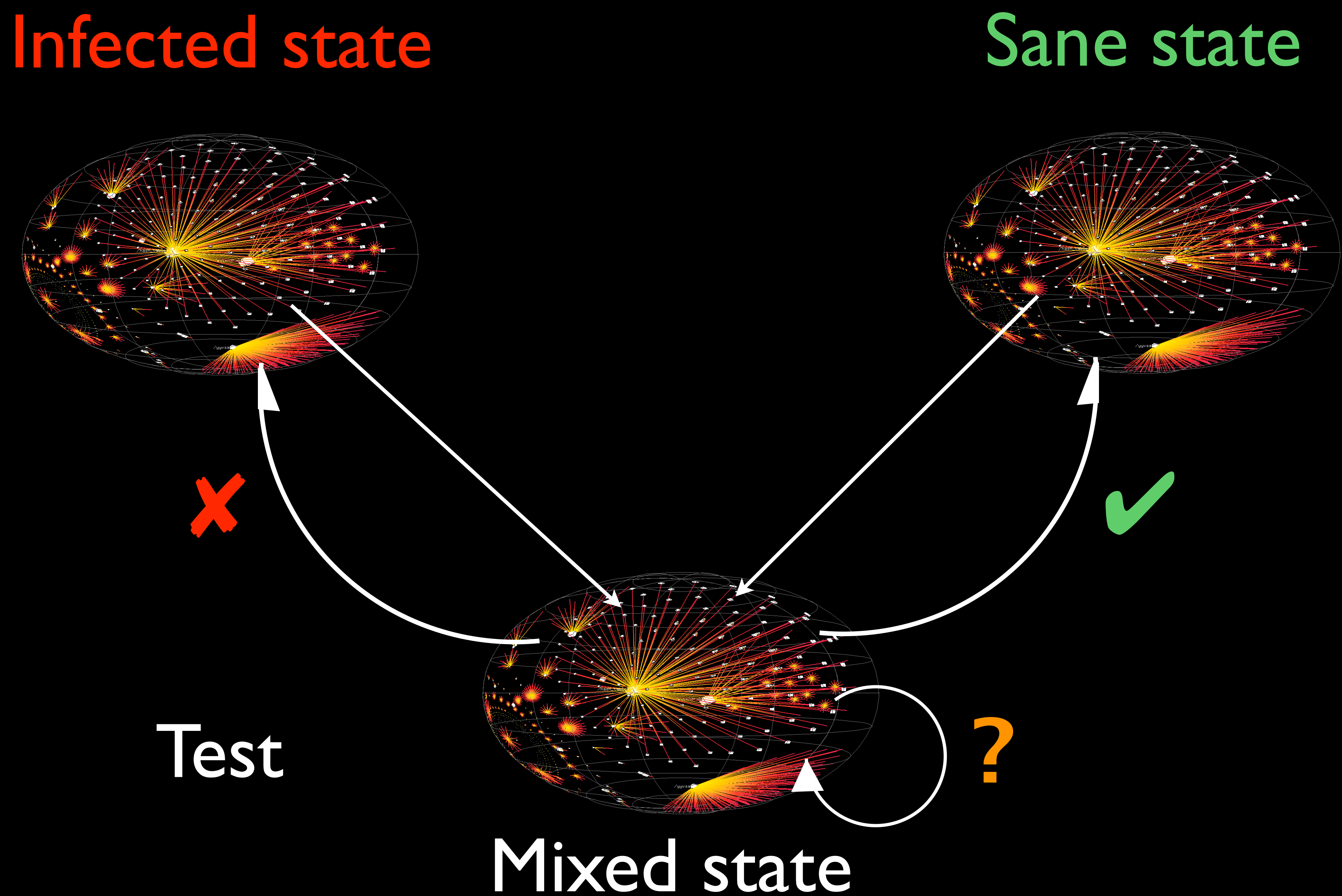
Infected state

Sane state

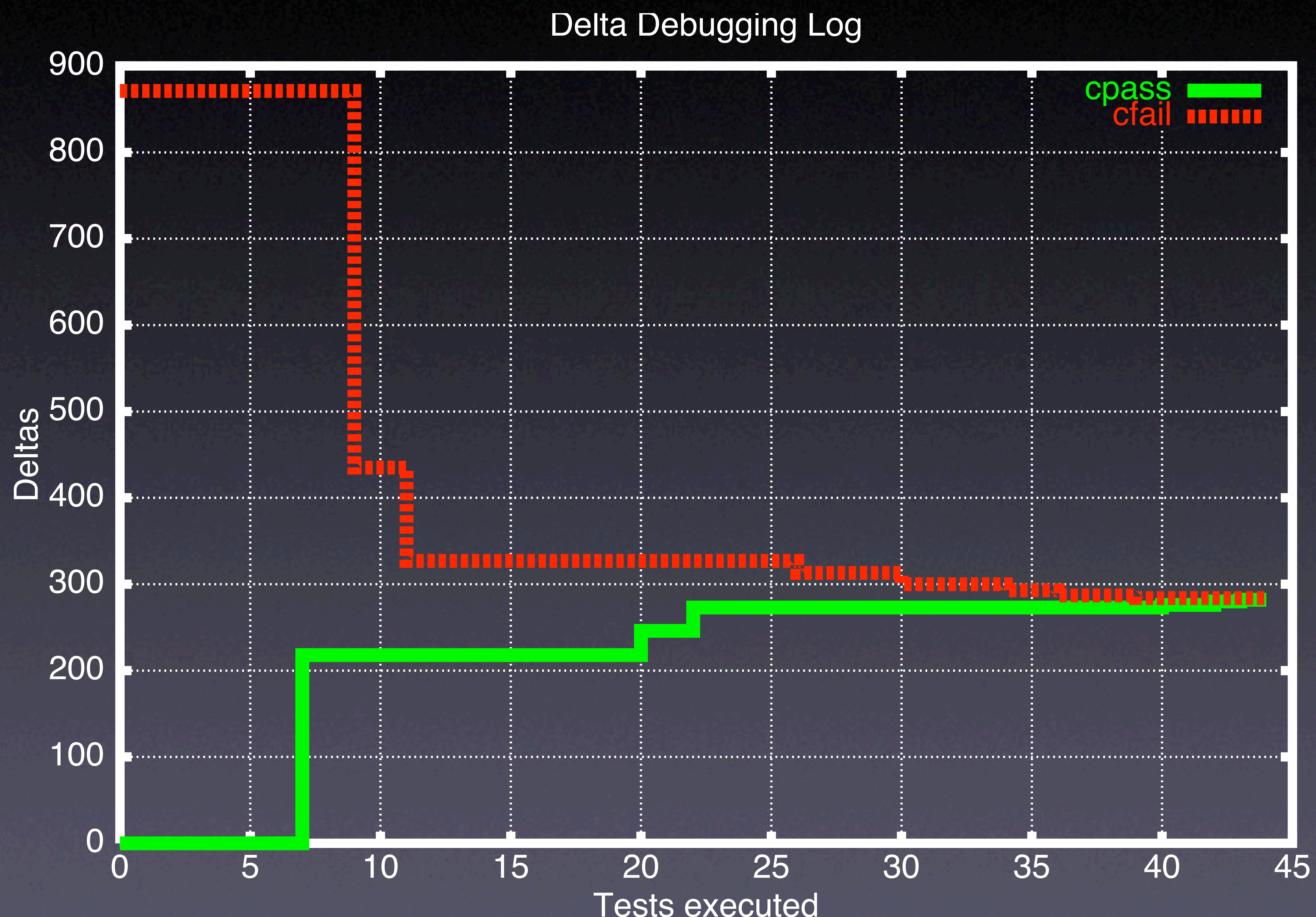


The difference
causes GCC to crash!

Search in Space



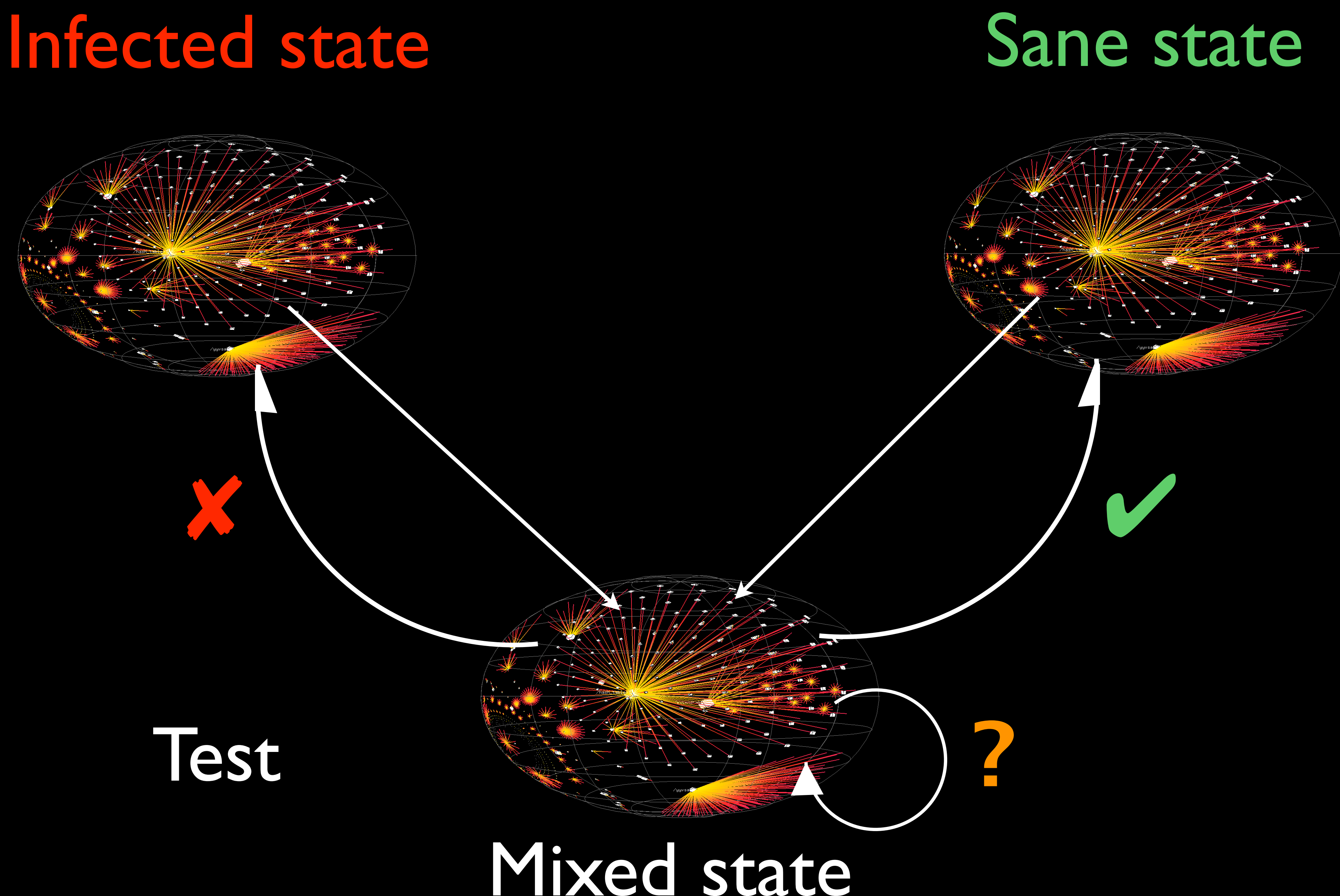
Search in Space



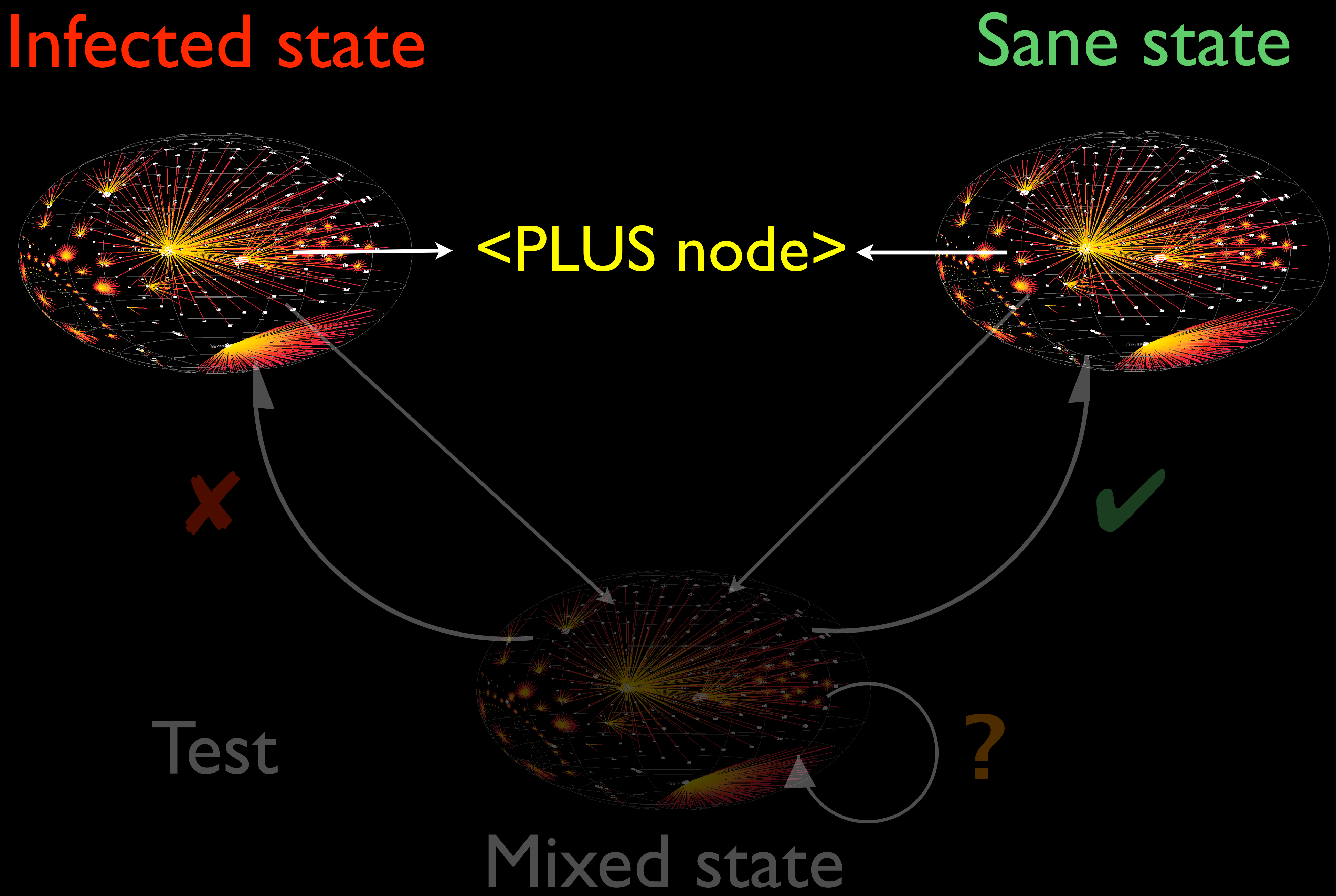
Search in Space

first_loop_store_insn→fld[1].rtx→fld[1].rtx→fld[3].rtx→fld[1].rtx→code == PLUS

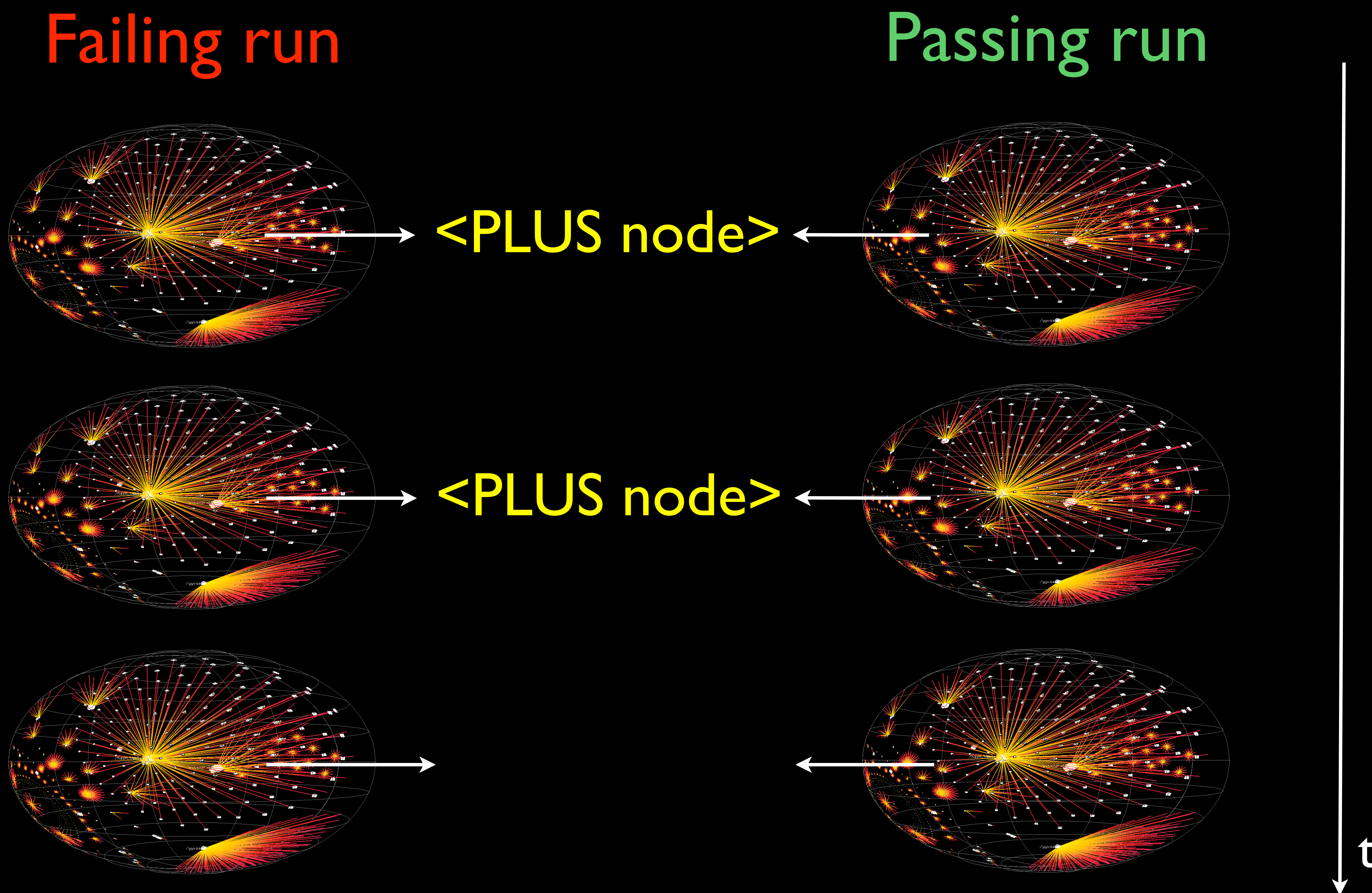
Search in Space



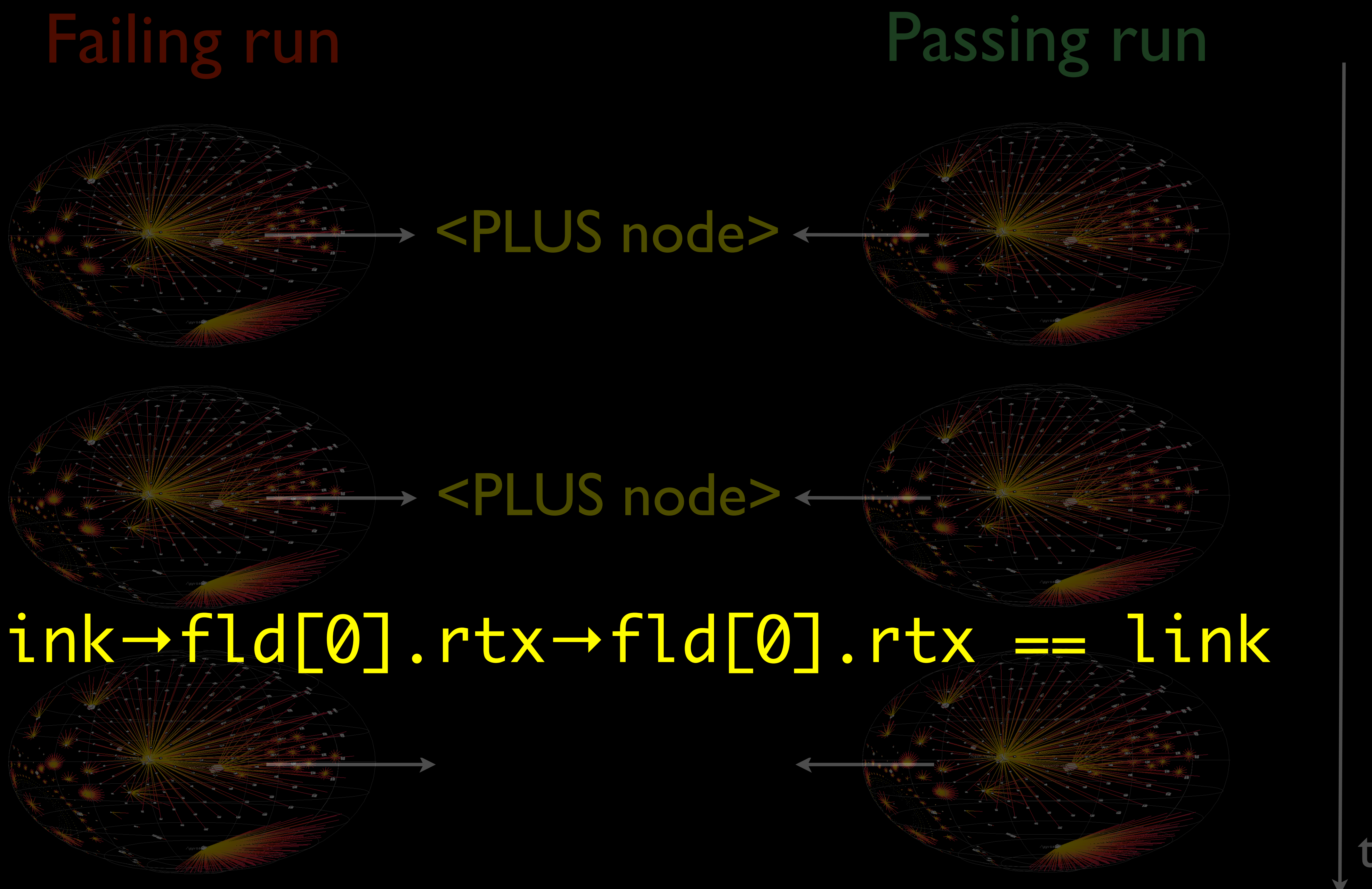
Search in Space



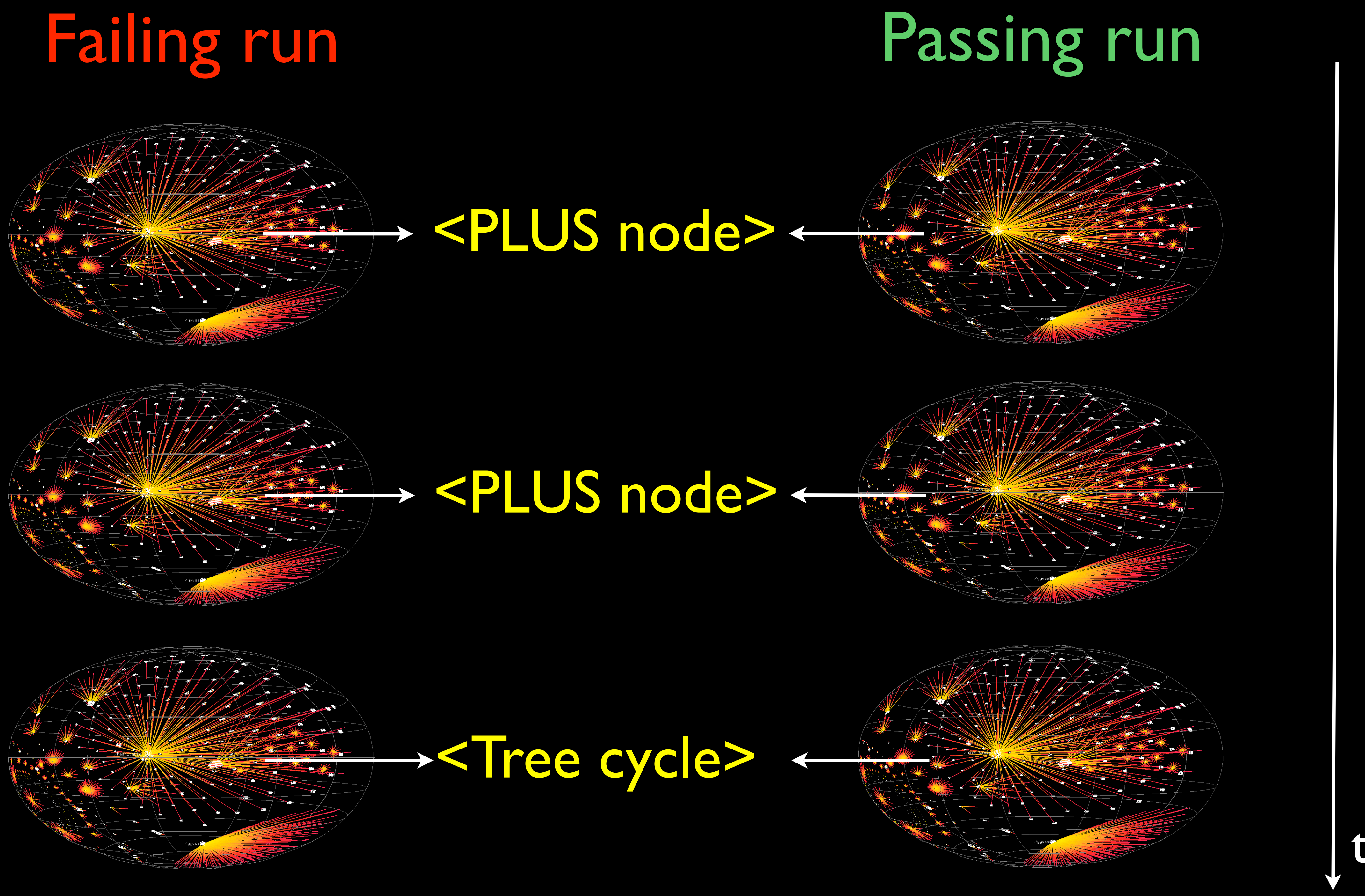
Search in Time

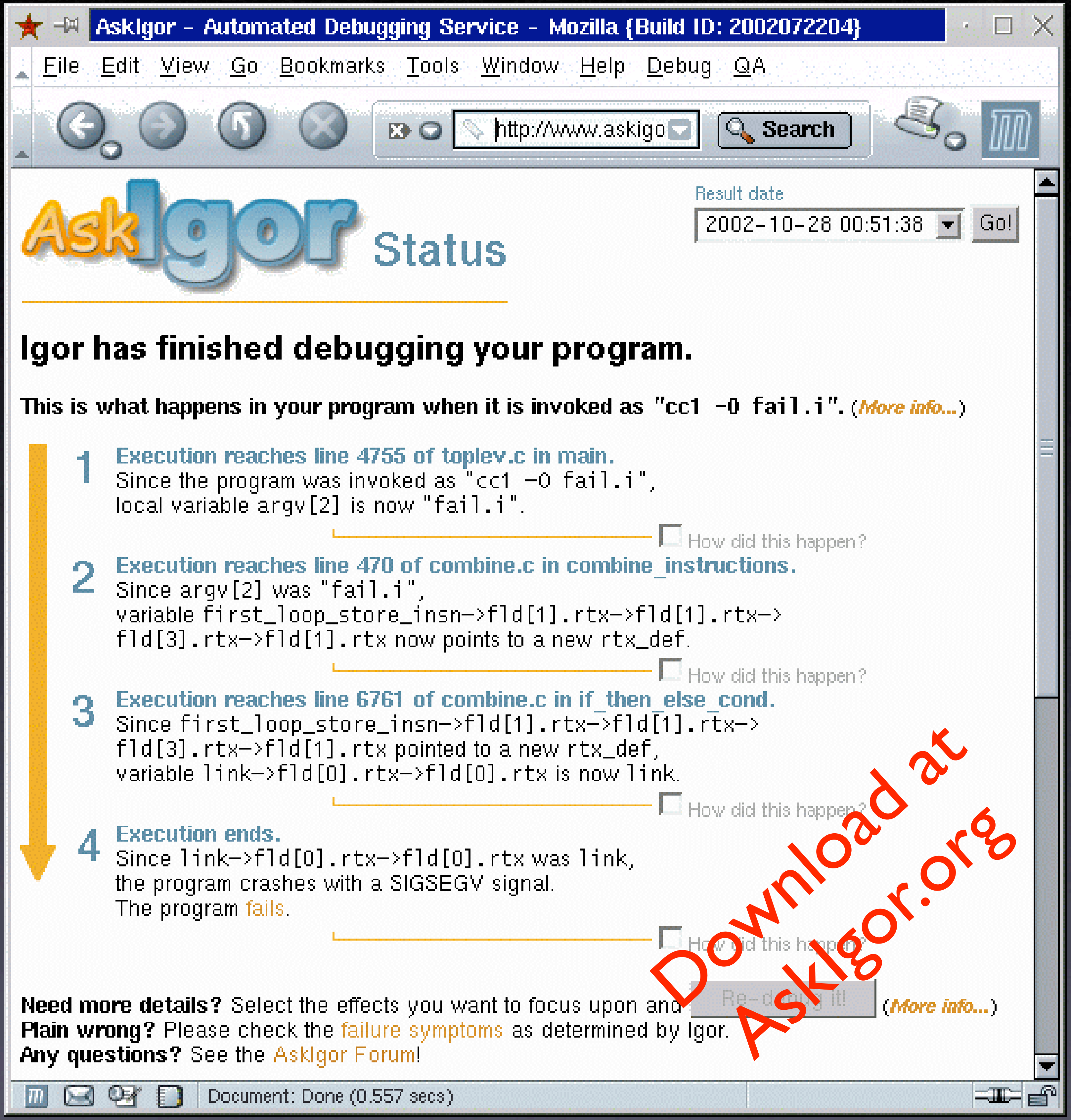


Search in Time



Search in Time




AskIgor - Automated Debugging Service - Mozilla {Build ID: 2002072204}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://www.askigo
Search
AskIgor Status
Result date
2002-10-28 00:51:38
Go!
Igor has finished debugging your program.
This is what happens in your program when it is invoked as "cc1 -O fail.i". (More info...)
1 Execution reaches line 4755 of toplev.c in main.
Since the program was invoked as "cc1 -O fail.i",
local variable argv[2] is now "fail.i".
How did this happen?
2 Execution reaches line 470 of combine.c in 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 now points to a new rtx_def.
How did this happen?
3 Execution reaches line 6761 of combine.c in if_then_else_cond.
Since first_loop_store_insn->fld[1].rtx->fld[1].rtx->
fld[3].rtx->fld[1].rtx pointed to a new rtx_def,
variable link->fld[0].rtx->fld[0].rtx is now link.
How did this happen?
4 Execution ends.
Since link->fld[0].rtx->fld[0].rtx was link,
the program crashes with a SIGSEGV signal.
The program fails.
Need more details? Select the effects you want to focus upon and Re-debug it! (More info...)
Plain wrong? Please check the failure symptoms as determined by Igor.
Any questions? See the AskIgor Forum!
Document: Done (0.557 secs)
Download at
AskIgor.org

Capturing State

for Python programs

```
if __name__ == "__main__":
    sys.settrace(tracer)
    ...

def tracer(frame, event, arg):
    dump_stack(frame)
    return tracer
```

Capturing State

for Python programs

```
def dump_stack(frame):
    while frame is not None:
        dump_frame(frame)
        frame = frame.f_back

def dump_frame(frame):
    locals = frame.f_locals
    globals = frame.f_globals
    print locals, globals
```

Manipulating State

for Python programs

```
def dump_frame(frame):
    locals = frame.f_locals
    locals['a'] = 42
```

equivalent to assignment
“a = 42” in frame

Caveats

Python frame objects are translated back to internal frames *only after tracer() has returned:*

- Frames can be *inspected* at any time, but *changed* only in tracer()
- Output of variables during tracer() may inhibit their translation at return

Open Issues

- How do we capture an accurate state?
- How do we ensure the cause is valid?
- Where does a state end?
- What is the cost?
- *When* do we compare states? (next lecture)

Concepts

- Delta Debugging on program states isolates a *cause-effect chain* through the run
- Use *memory graphs* to extract and compare program states
- Demanding, yet effective technique