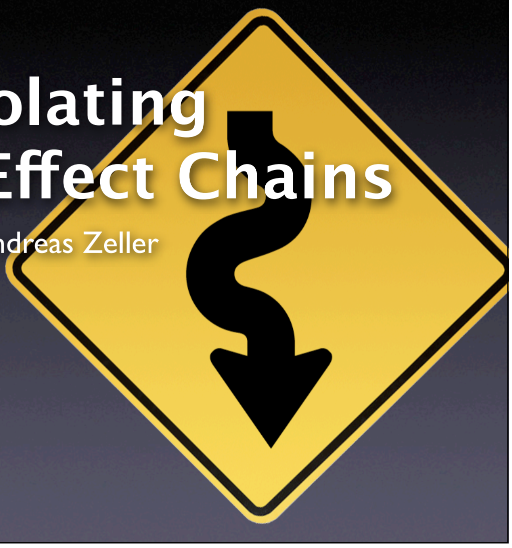
Isolating Cause–Effect Chains

Andreas Zeller



1

bug.c

```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];
}
```


2

What is the cause
of this failure?

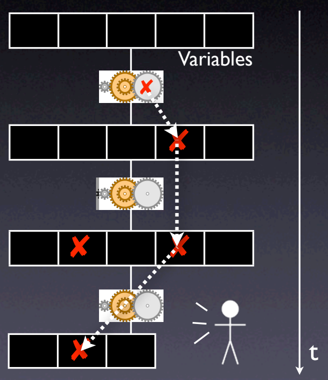
What do we do now?

3

From Defect to Failure

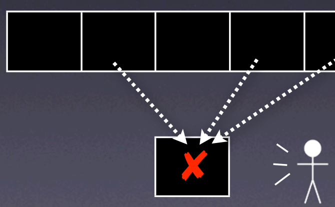
1. The programmer creates a *defect* – an error in the code.
2. When executed, the defect creates an *infection* – an error in the state.
3. The infection *propagates*.
4. The infection causes a *failure*.

This infection chain must be traced back – and broken.

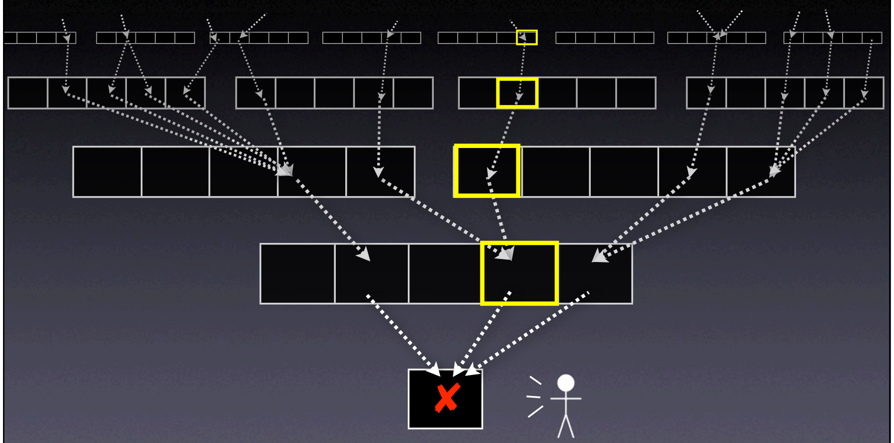


Tracing Infections

- 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

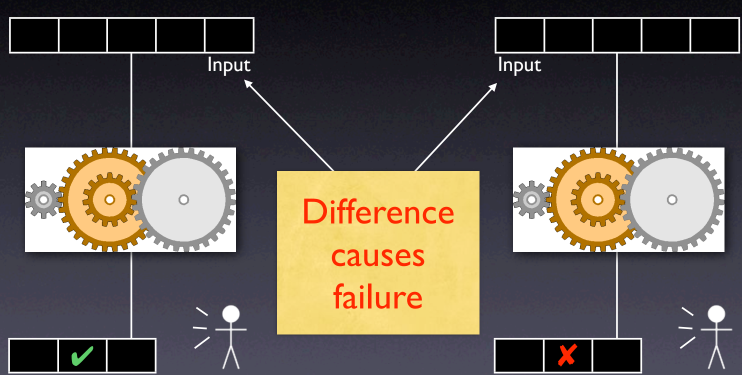
Obscure Code

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

tp	len	p[0]

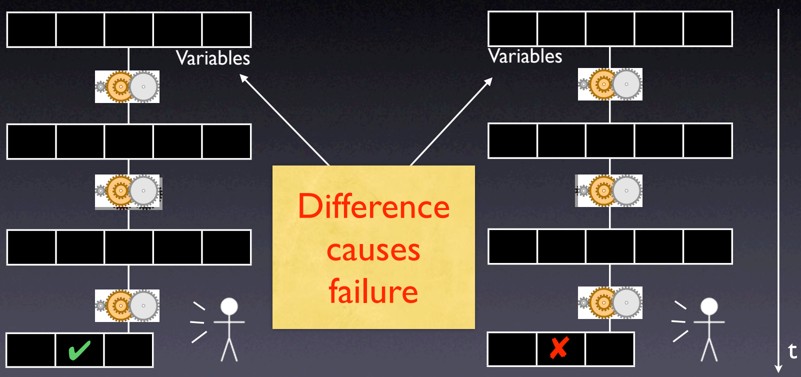
And even if we know everything, there still is code which is almost impossible to analyze. In C, for instance, only the programmer knows how memory is structured; there is no general way for static analysis to find this out

Isolating Input



In the last lecture, we have seen delta debugging on input.

Isolating States



Now let's take a deeper view. If a program is a succession of states, can't we treat each state as **an input to the remainder of the run?**

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?

Let's look at a simpler example first.

```
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 $r_{\checkmark}$	Value in $r_{\times}$
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 $r_{\checkmark}$	Value in $r_{\times}$
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					■									

16

16

17

Since this worked so well, we built a debugging server.

17

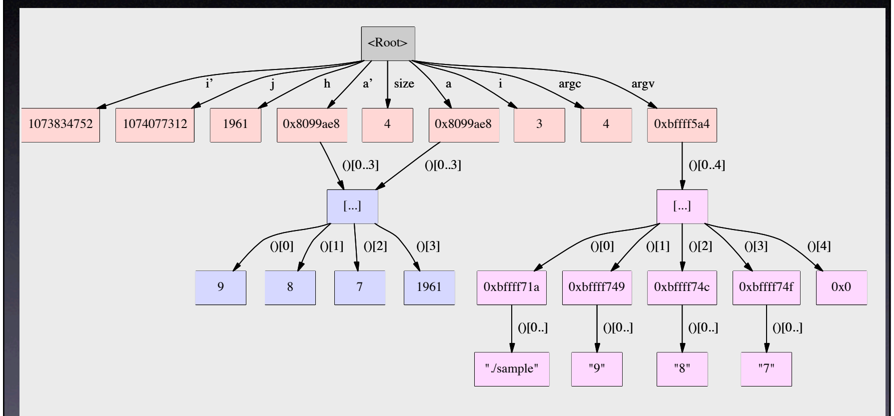
18

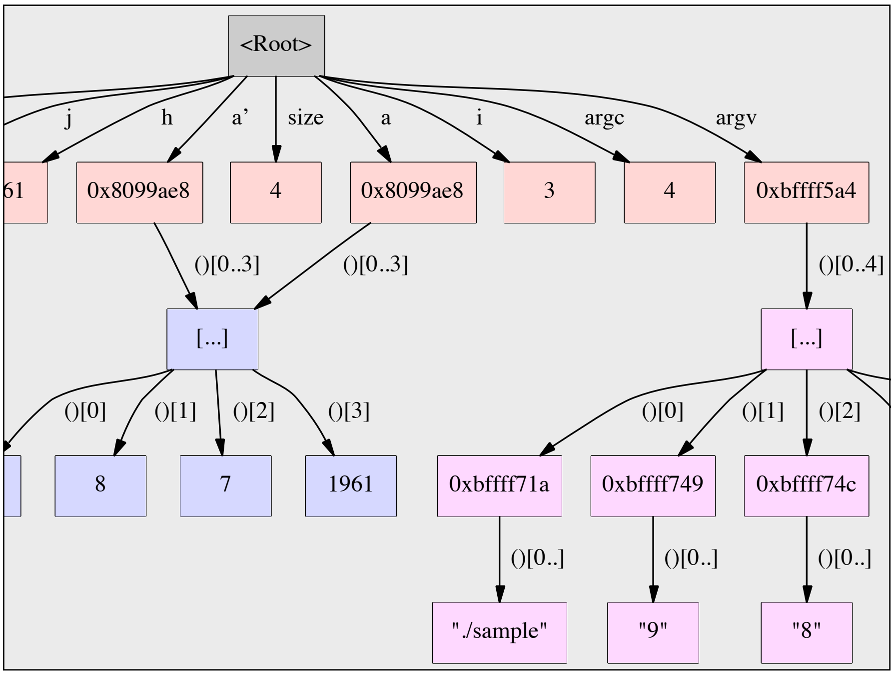
18

Complex State

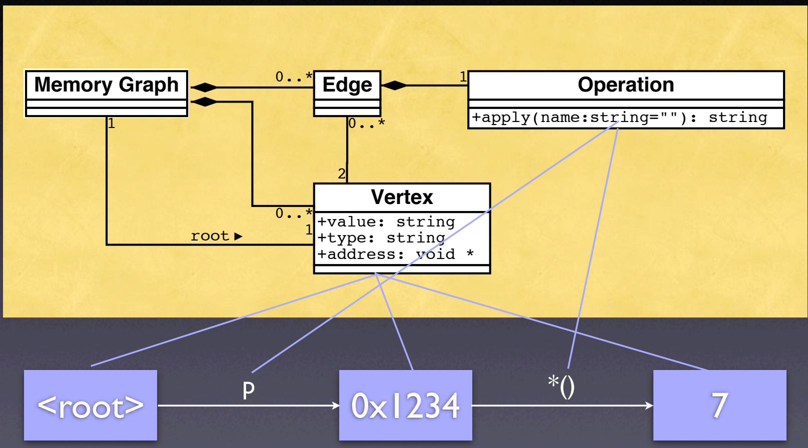
- Accessing the state as a *table* is not enough:
  - References are not handled
  - Aliases are not handled
- We need a *richer* representation

A Memory Graph





Structure



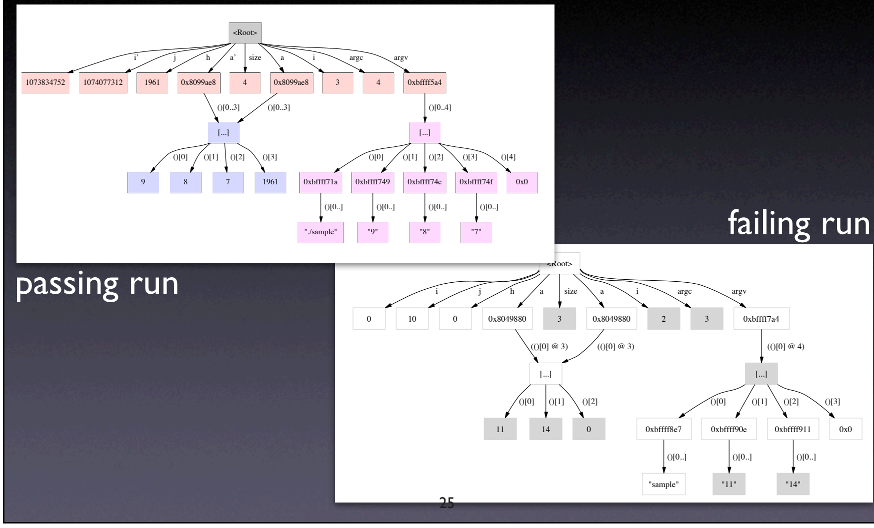
Construction

- 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

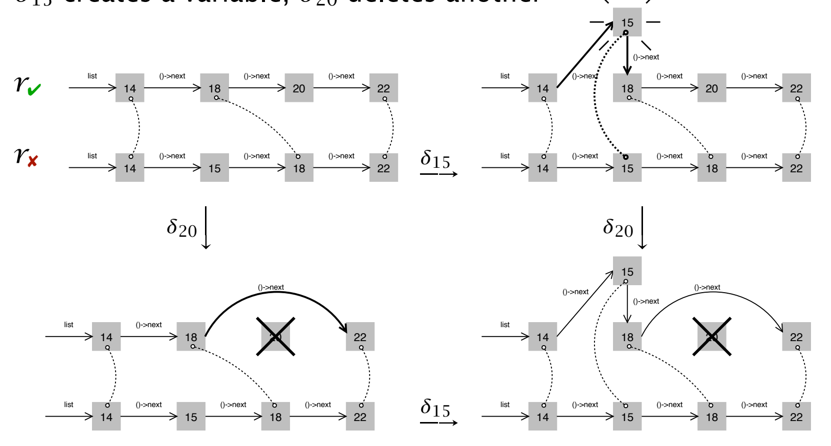


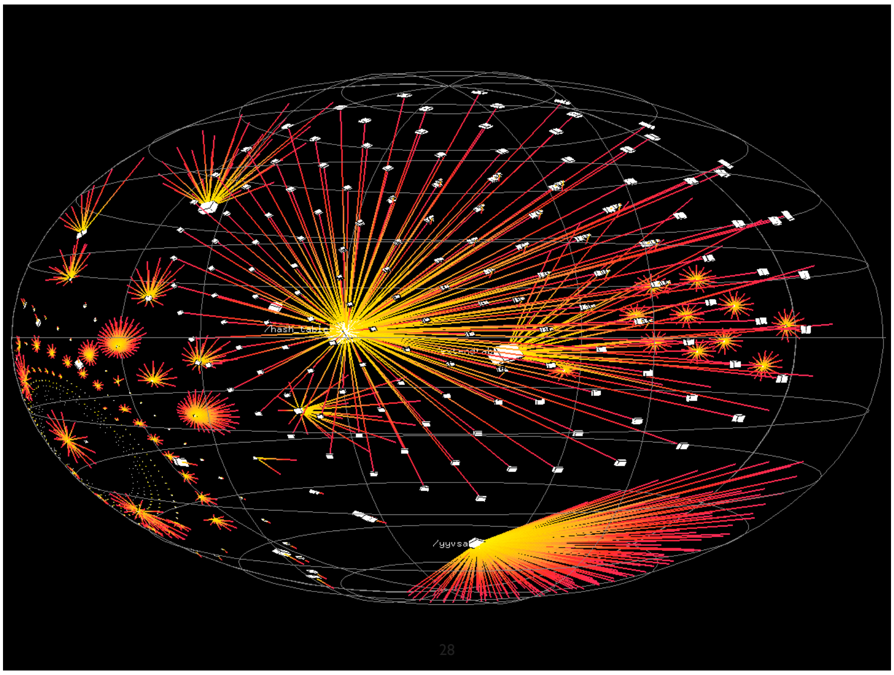
Comparing States

- 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

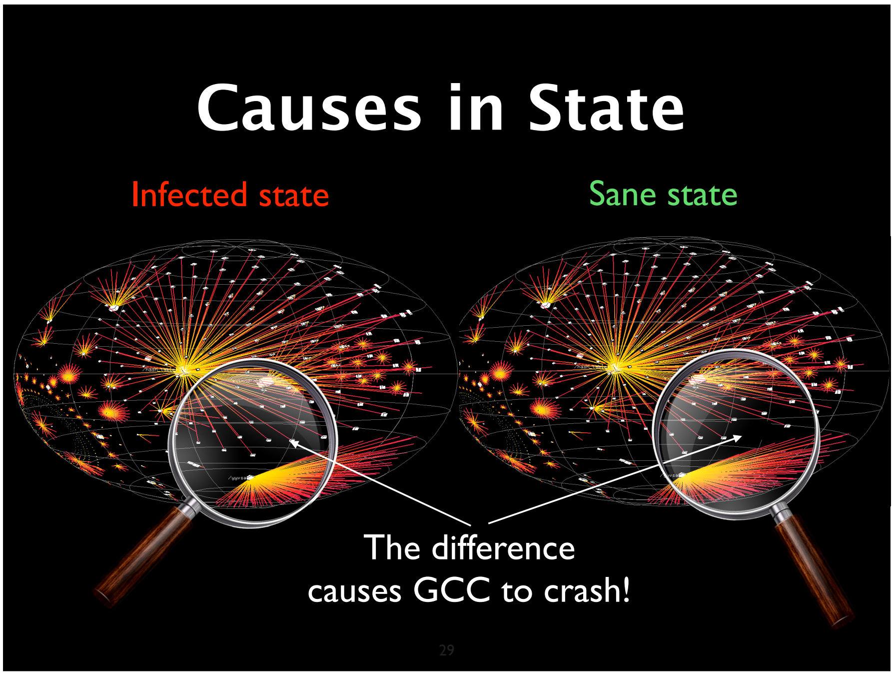
δ_{15} creates a variable, δ_{20} deletes another



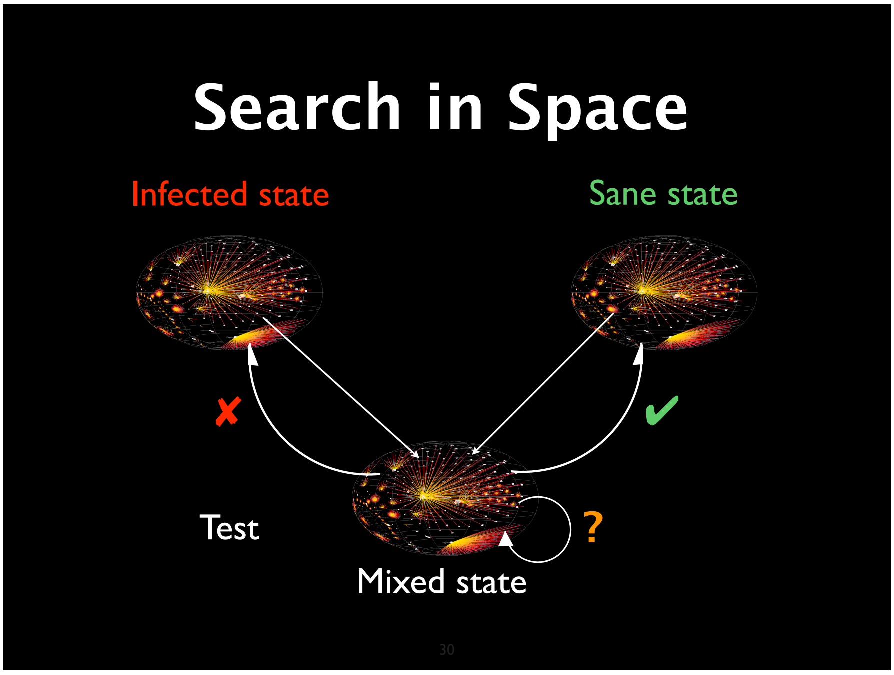


State of the GNU compiler (GCC)
42991 vertices
44290 edges - and 1 is wrong :-)
An actual GCC execution has millions of these states.

28

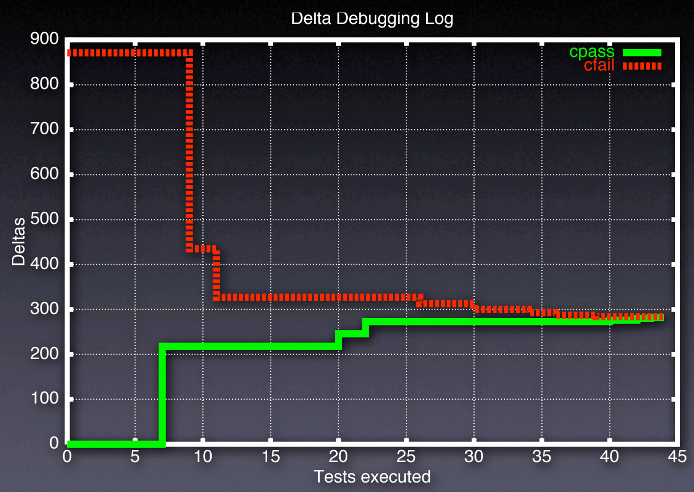


29



30

Search in Space



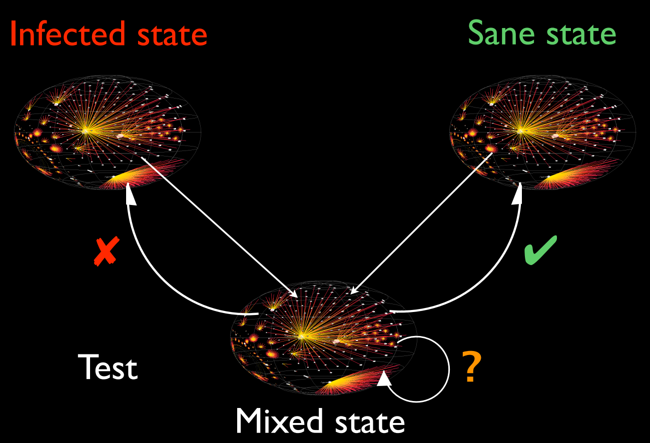
31

Search in Space

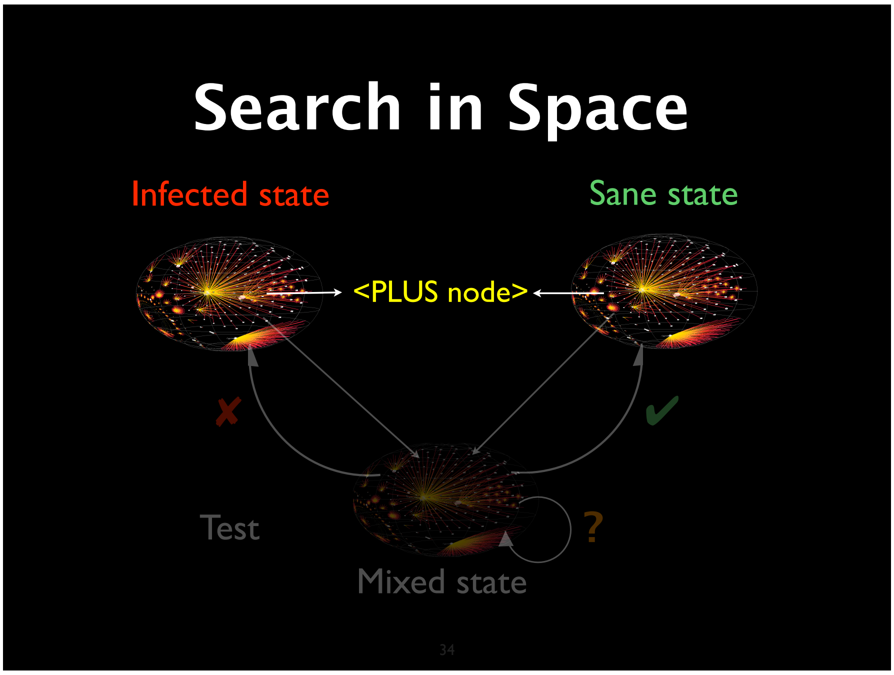
first_loop_store_insn→fld[1].rtx→fld[1].rtx→fld[3].rtx→fld[1].rtx→code == PLUS

32

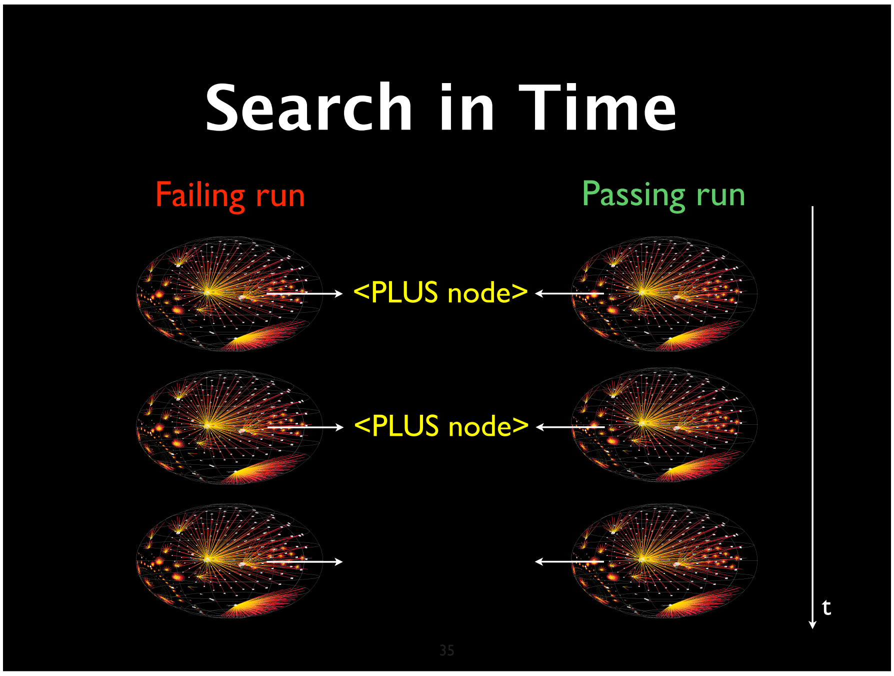
Search in Space



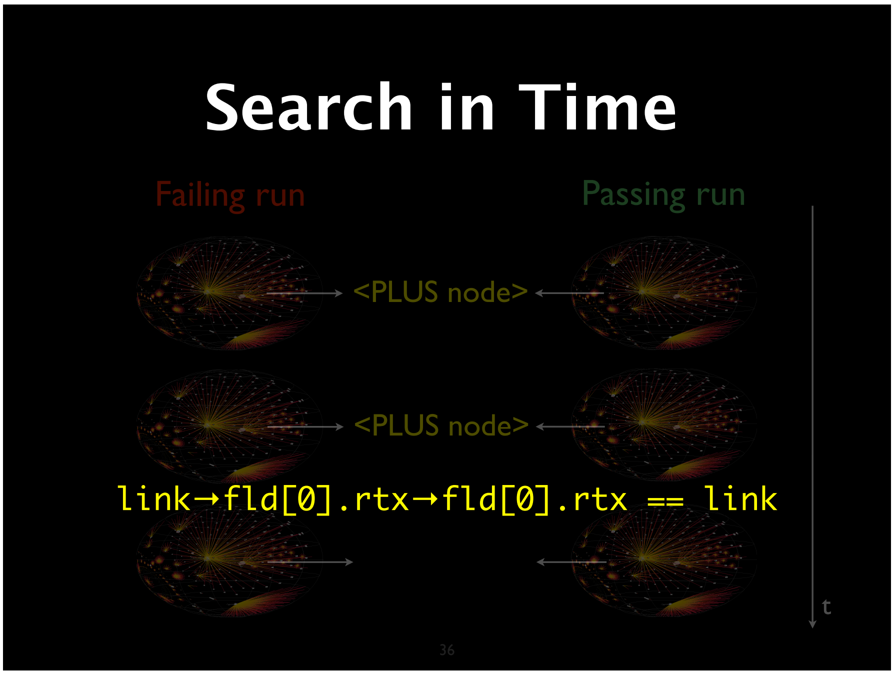
33


Search in Space
Infected state
Sane state
<PLUS node>
Test
?
Mixed state
34

34

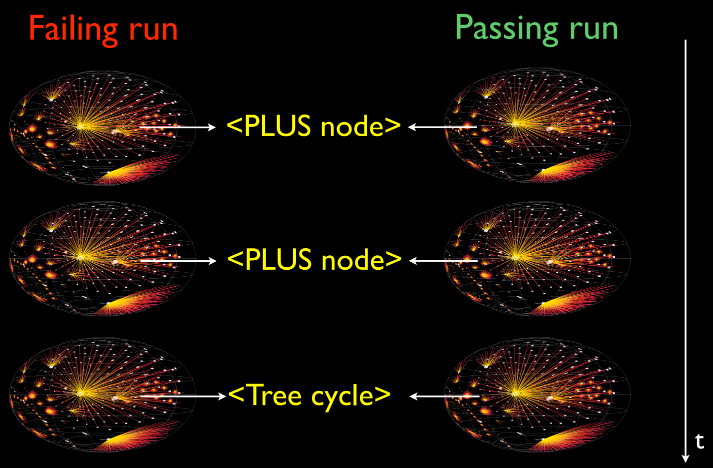

Search in Time
Failing run
Passing run
<PLUS node>
<PLUS node>
t
35

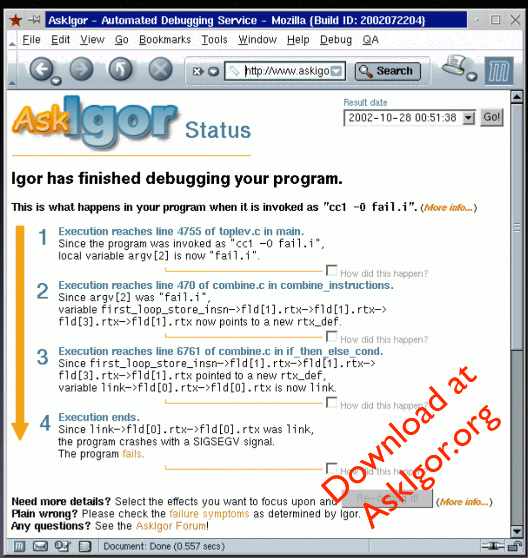
35


Search in Time
Failing run
Passing run
<PLUS node>
<PLUS node>
link→fld[0].rtx→fld[0].rtx == link
t
36

36

Search in Time





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
```

40

40

Manipulating State

for Python programs

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

equivalent to assignment
"a = 42" in frame

41

41

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

42

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