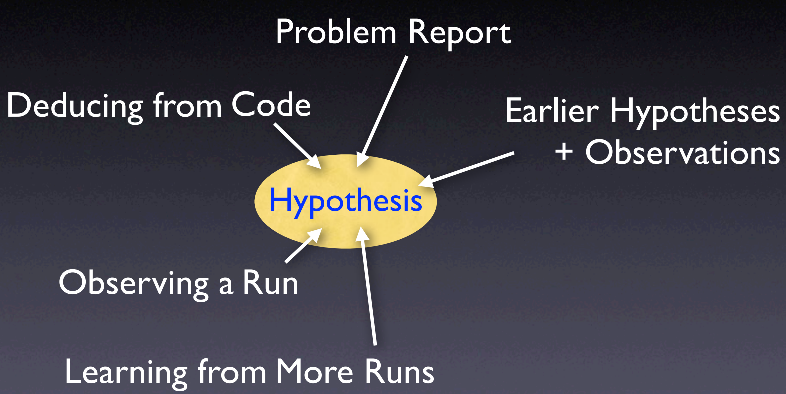
Deducing Errors

Andreas Zeller

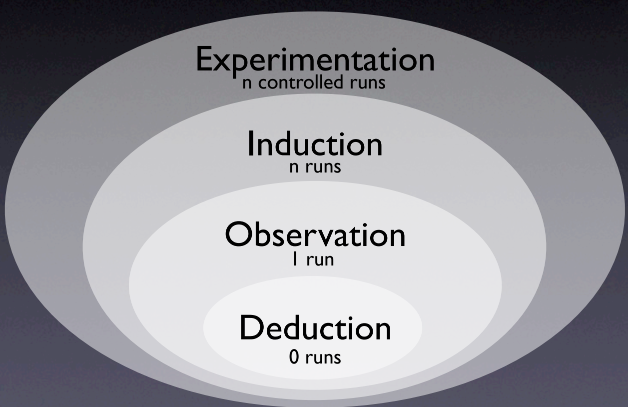
1

Obtaining a Hypothesis



2

Reasoning about Runs



3

Reasoning about Runs

Deduction
0 runs

4

What's relevant?

```
10 INPUT X
20 Y = 0
30 X = Y
40 PRINT "X = ", X
```


5

Fibonacci Numbers

$$fib(n) = \begin{cases} 1, & \text{for } n = 0 \lor n = 1 \\ fib(n-1) + fib(n-2), & \text{otherwise} \end{cases} .$$

1	1	2	3	5	8	13	21	34	55

6

fibo.c

```
int fib(int n)
{
    int f, f0 = 1, f1 = 1;

    while (n > 1) {
      n = n - 1;
      f = f0 + f1;
      f0 = f1;
      f1 = f;
    }

    return f;
}
```

```
int main()
{
  int n = 9;

  while (n > 0)
  {
    printf("fib(%d)=%d\n",
           n, fib(n));
    n = n - 1;
  }

    return 0;
}
```


Fibo in Action

```
$ gcc -o fibo fibo.c
$ ./fibo
fib(9)=55
fib(8)=34
...
fib(2)=2
fib(1)=134513905
```

Where does fib(1) come from?

Effects of Statements

- Write. A statement can change the program state (i.e. write to a variable)
- Control. A statement may determine which statement is executed next (other than unconditional transfer)

Affected Statements

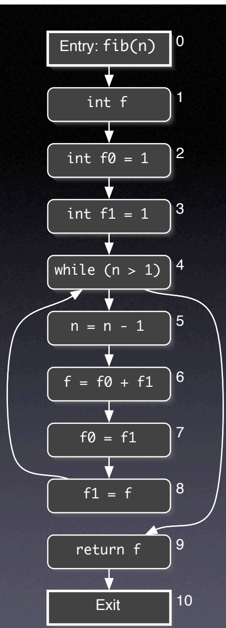
- Read. A statement can read the program state (i.e. from a variable)
- Execution. To have any effect, a statement must be executed.

Effects in fibo.c

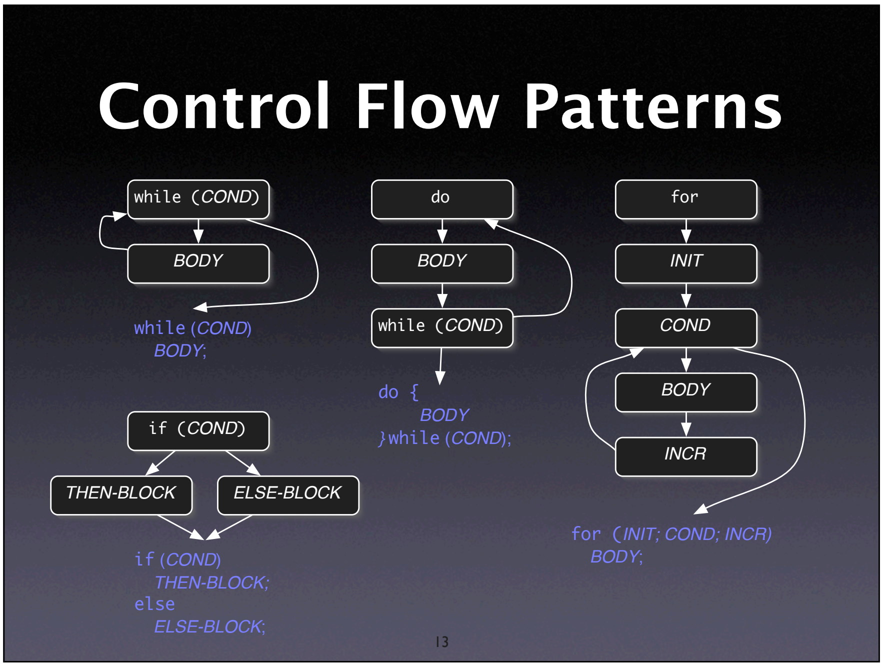
	Statement	Reads	Writes	Controls
0	`fib(n)`		n	1-10
1	`int f`		f	
2	`f0 = 1`		f0	
3	`f1 = 1`		f1	
4	`while (n > 1)`	n		5-8
5	`n = n - 1`	n	n	
6	`f = f0 + f1`	f0, f1	f	
7	`f0 = f1`	f1	f0	
8	`f1 = f`	f	f1	
9	`return f`	f	<ret>	

Control Flow

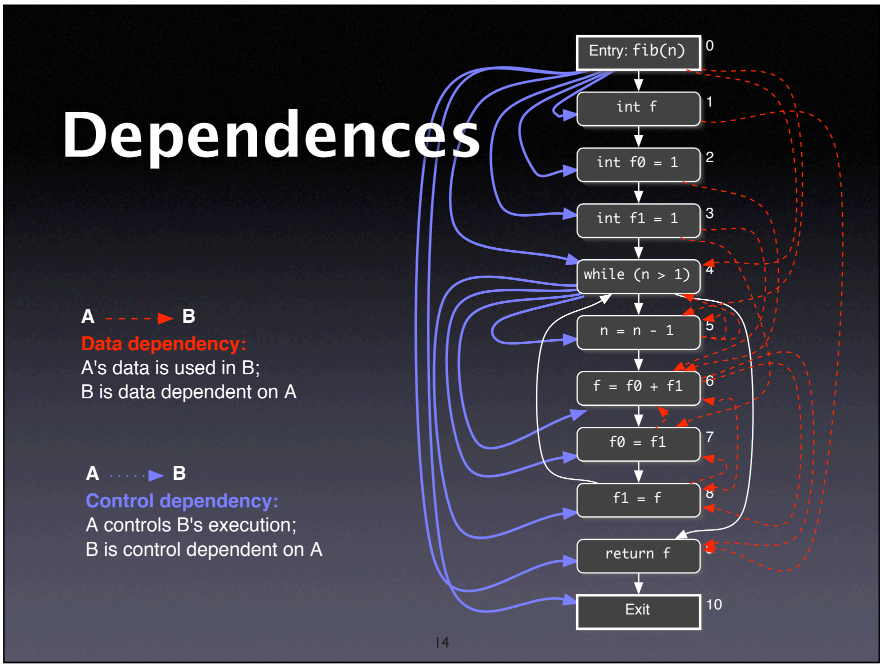
```
int fib(int n)
{
    int f, f0 = 1, f1 = 1;

    while (n > 1) {
      n = n - 1;
      f = f0 + f1;
      f0 = f1;
      f1 = f;
    }

    return f;
}
```



The CFG is best developed incrementally on an extra board.

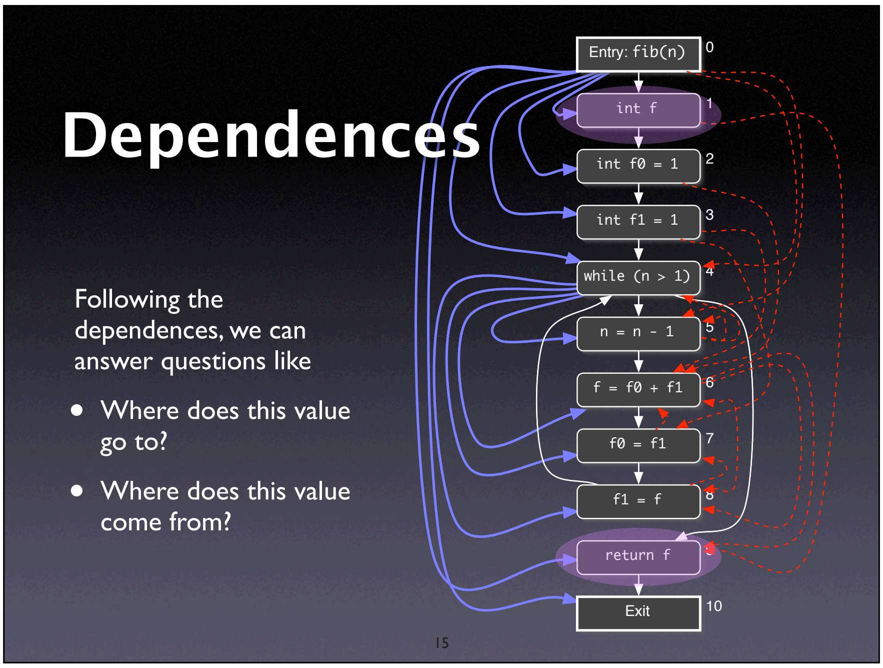


13



14

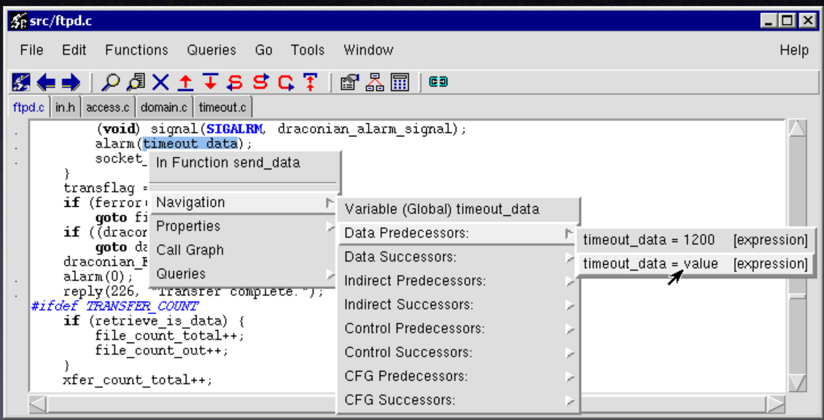
Again, this is best developed interactively on the board (possibly by having the students call further dependences)



15

Again, this is best developed interactively on the board (possibly by having the students call further dependences)

Navigating along Dependences

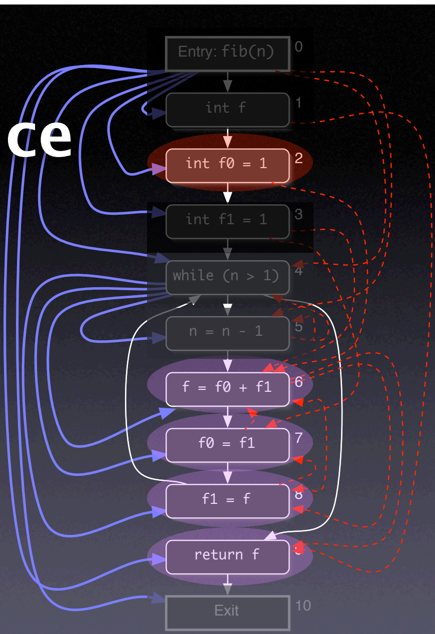


Program Slicing

- A *slice* is a subset of the program
- Allows programmers to *focus on what's relevant* with respect to some statement S:
  - All statements influenced by S
  - All statements that influence S

Forward Slice

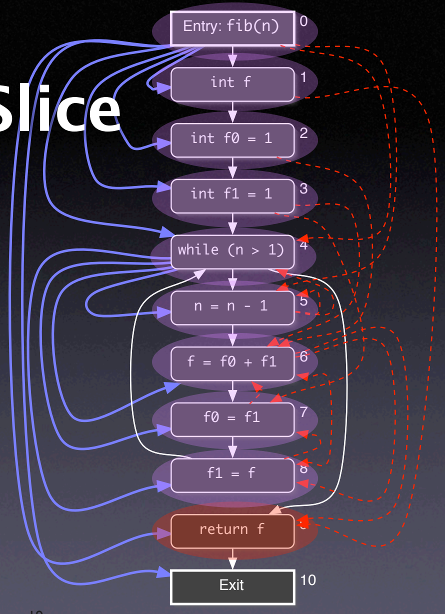
- Given a statement A, the forward slice contains all statements whose read variables or execution could be influenced by A
- Formally:
  $S^F(A) = \{B \mid A \rightarrow^* B\}$



Again, this is best developed interactively on the board (possibly by having the students call further dependences)

Backward Slice

- Given a statement B, the backward slice contains all statements that could influence the read variables or execution of B
- Formally:
  $S^B(B) = \{A \mid A \to^* B\}$



19

Again, this is best developed interactively on the board (possibly by having the students call further dependences)

19

Two Slices

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Slice Operations:

- Backbones
- Dices
- Chops

Backward slice of sum

Backward slice of mul

20

20

Backbone

```
  a = read();
  b = read();
  while (a <= b) {

    a = a + 1;
```

- Contains only those statement that occur in both slices
- Useful for focusing on common behavior

21

21

Two Slices

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);    <- Backward slice of sum
  write(mul);    <- Backward slice of mul
}
```

Slice Operations:

- Backbones
- Dices
- Chops

22

22

Dice

```
sum = 0;

  sum = sum + a;

write(sum);
```

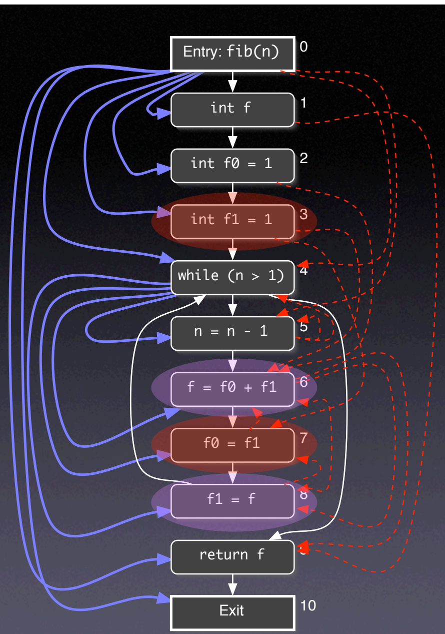
- Contains only the difference between two slices
- Useful for focusing on differing behavior

23

23

Chop

- Intersection between a forward and a backward slice
- Useful for determining influence paths within the program

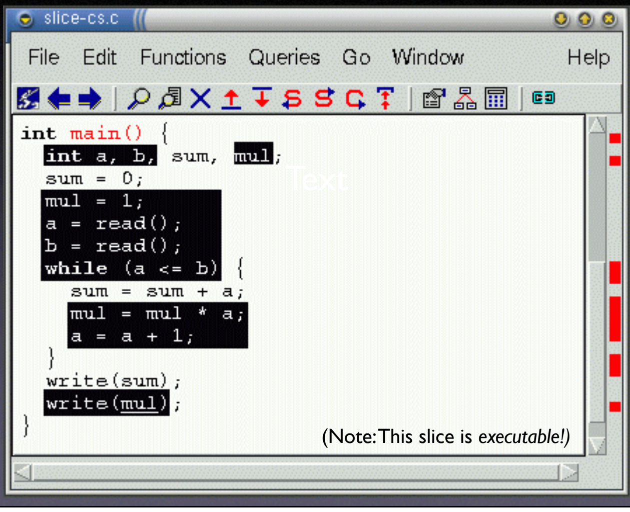


24

24

Again, this is best developed interactively on the board (possibly by having the students call further dependences)

Leveraging Slices



25

Deducing Code Smells

- Use of uninitialized variables
- Unused values
- Unreachable code
- Memory leaks
- Interface misuse
- Null pointers

26

26

Uninitialized Variables

```
$ gcc -Wall -O -o fibo fibo.c
fibo.c: In function `fib':
fibo.c:7: warning: `f' might be
used uninitialized in this
function
```

27

27

False Positives

```
int go;
switch (color) {
    case RED:
    case AMBER:
        go = 0;
        break;
    case GREEN:
        go = 1;
        break;
}
if (go) { ... }
```

warning: `go' might be used uninitialized in this function

28

Unreachable Code

```
if (w >= 0)
    printf("w is non-negative\n");
else if (w > 0)
    printf("w is positive\n");
```

warning: will never be executed

29

Memory Leaks

```
int *readbuf(int size)
{
    int *p = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        p[i] = readint();
        if (p[i] == 0)
            return 0;  // end-of-file
    }
    return p;
}
```

memory leak

30

Interface Misuse

```
void readfile()
{
    int fp = open(file);
    int size = readint(file);
    if (size <= 0)
        return;
    ...
    close(fp);
}
```

stream not closed

31

31

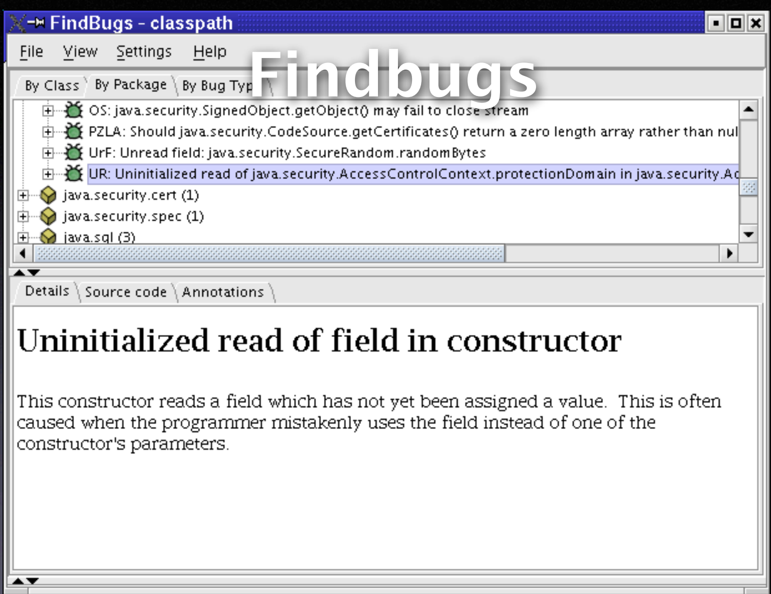
Null Pointers

```
int *readbuf(int size)
{
    int *p = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        p[i] = readint();
        if (p[i] == 0)
            return 0;   // end-of-file
    }
    return p;
}
```

p may be null

32

32



33

33

Defect Patterns

- Class implements `Cloneable` but does not define or use `clone` method
- Method might ignore exception
- Null pointer dereference in method
- Class defines `equal()`; should it be `equals()`?
- Method may fail to close database resource
- Method may fail to close stream
- Method ignores return value
- Unread field
- Unused field

Limits of Analysis

```
int x;
for(i=j=k=1;--j||k;k=j?i%j?k:k-j:(j=i+=2));
write(x);
```

- Is x being used uninitialized or not?
- Loop halts only if there is an odd perfect number (= a number that’s the sum of its proper positive divisors)
- Problem is undediced yet


```
static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
```

Conservative approximation: any a[] depends on all a[]

Causes of Imprecision

- Indirect access, as in a[i]
- Pointers
- Functions
- Dynamic dispatch
- Concurrency

37

37

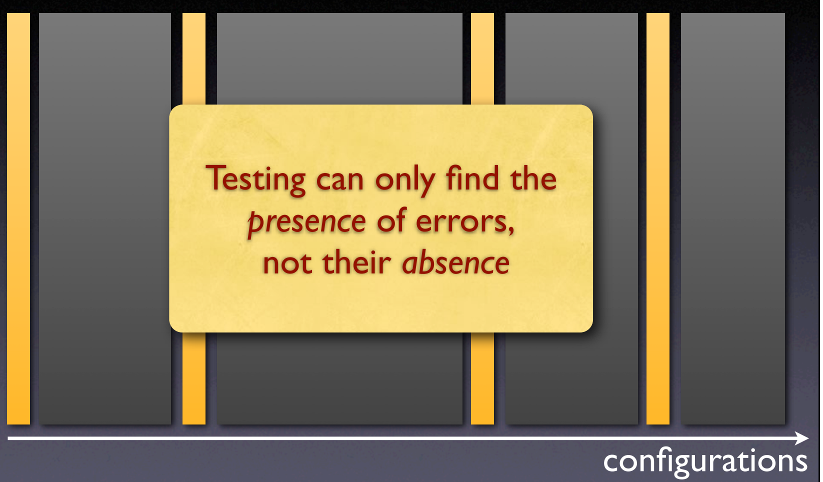
Risks of Deduction

- Code mismatch. Is the run created from this very source code?
- Imprecision. A slice typically encompasses 90% of the source code.
- Abstracting away. Failures may be caused by a defect in the environment.

38

38

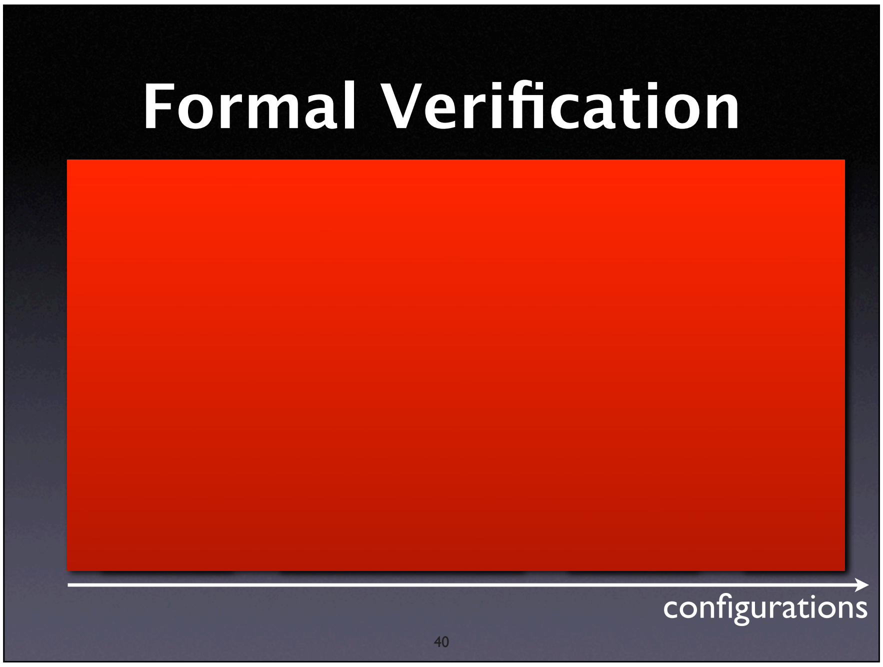
Dijkstra's Curse



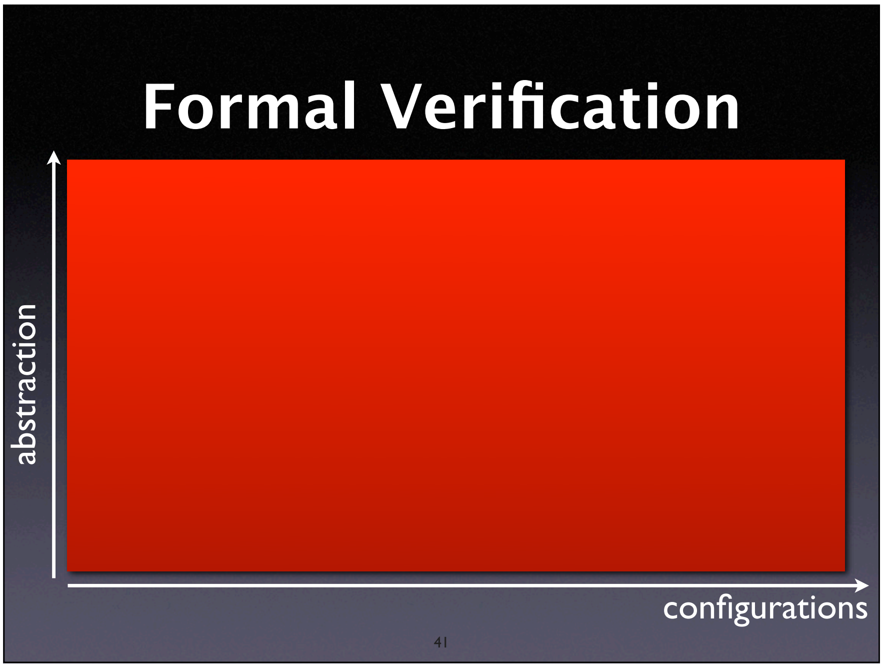
39

39

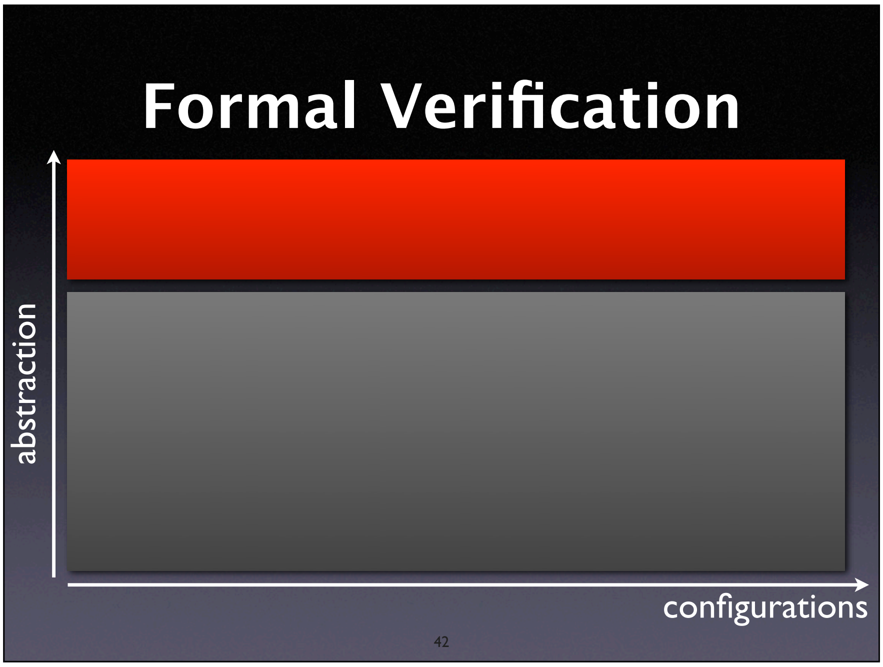
But still, testing suffers from what I call Dijkstra's curse – a double meaning, as it applies both to testing as to his famous quote. Is there something that can find the **absence** of errors?



40



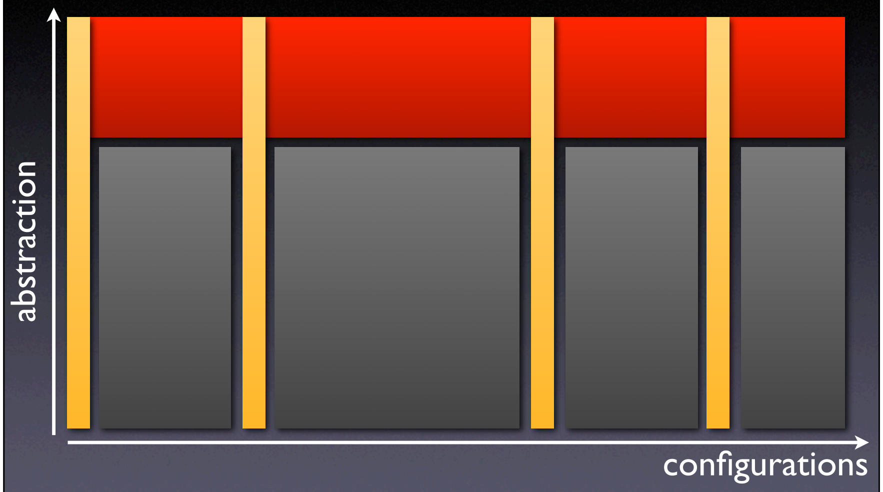
41



Areas missing might be: the operating system, the hardware, all of the world the system is embedded in (including humans!)

42

Best of Both Worlds



We might not be able to cover **all** abstraction levels in **all** configurations, but we can do our best to cover as much as possible.

Hetzel–Myers Law

A combination of different V&V methods outperforms any single method alone.

Increasing Precision

- Verification. If we know that certain properties hold, we can leverage them in our inference process.
- Observation. Facts from concrete runscan be combined with deduction.

…in the weeks to come!

Concepts

- ★ To reason about programs, use
  - deduction (0 runs)
  - observation (1 run)
  - induction (multiple runs)
  - experimentation (controlled runs)

46

46

Concepts (2)

- ★ To isolate value origins, follow back the dependences
- ★ Dependences can uncover *code smells* such as
  - uninitialized variables
  - unused values
  - unreachable code
- ★ Get rid of smells before debugging

47

47

Concepts (3)

- ★ To slice a program, follow dependences from a statement S to find all statements that
  - could be influenced by S (forward slice)
  - could influence S (backward slice)

48

48

Concepts (4)

- ★ Using deduction alone includes a number of risks, including code mismatch, sbstracting away, and imprecision.
- ★ Any deduction is limited by the halting problem and must thus resort to conservative approximation.
- ★ For debugging, deduction is best combined with actual observation.