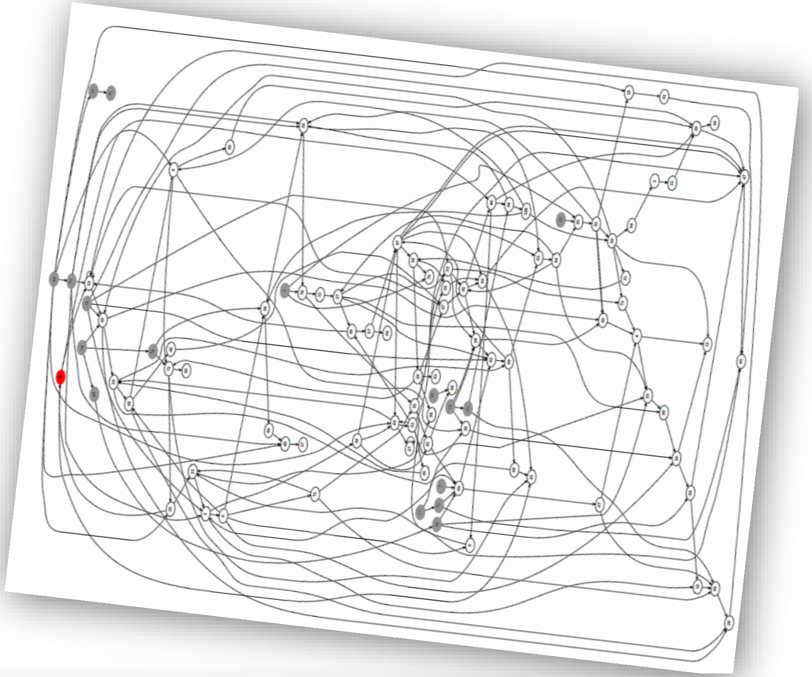
WHAT'S THE FUZZ ABOUT (SOFTWARE) TESTING?

Gerard Holzmann

Nimble Research

gholzmann@acm.org

MC/DC TESTING OF CRITICAL SOFTWARE



ISO 26262: highly recommended
EN 50128: highly recommended
IEC 61508: highly recommended
DO 178C: required

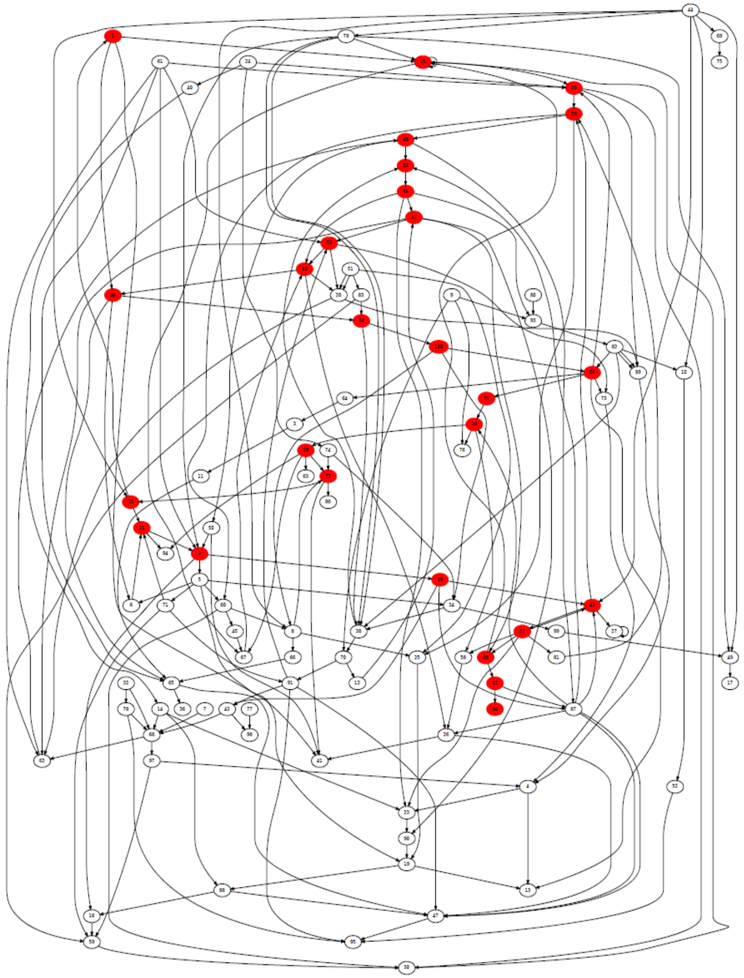
Modified condition/decision coverage – Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect th

as opposed to testing only expected behavior,
or randomly poking the code with inputs

“Whatever can happen will happen if we make trials enough.”
Augustus De Morgan (1866)

QUESTIONS

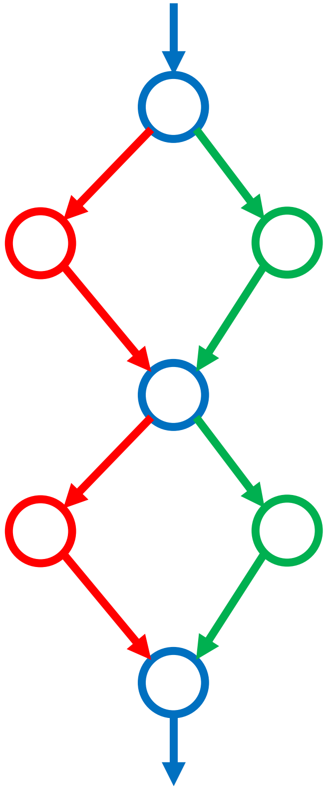
1. How good is Software Testing with 100% *MC/DC* Coverage ?
2. Is *Randomized* Testing *(Fuzz testing)* better ?
3. Does it change if we *Remember* Nodes we’ve visited ? (using *Perfect Recall)*
4. Can we use *Parallelism* to speed things up if all this starts taking too much time ?



SOME EXAMPLES
1: TESTING CONDITIONAL CODE

```
int *p;

void
fct(int x, int y)
{
    if (x)
    {    p = &x;
    }
    if (y)
    {   *p = y;
    }
}
```



```
void
test_main(void)
{
    fct(0,0);
    fct(1,1);
}
```

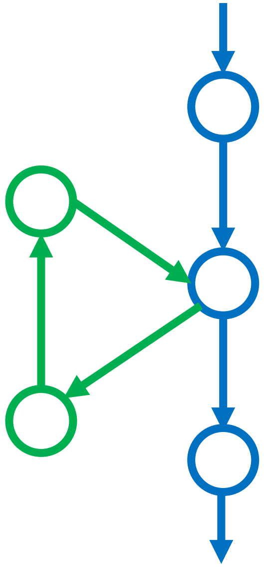
this test achieves 100% MC/DC coverage, yet it misses a serious bug that could be revealed with a third test: foo(0,1)

the MC/DC test covered just 50% of the paths in the control-flow graph

2: TESTING CODE LOOPS

```
void
fct(int x, int y)
{   int i, a[4];

    for (i = 0; i < x+y; i++)
    {   a[i] = i;
    }
}
```



```
void
test_main(void)
{
    fct(1,1);
}
```

this single test achieves 100% MC/DC coverage, but misses the array indexing bug that can be revealed with, for instance, foo(1,3)

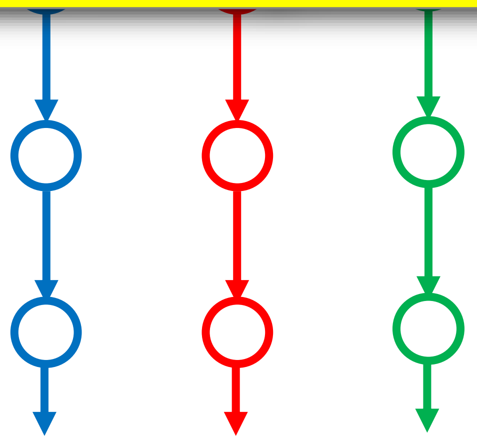
this 1 test covers just 1 of 2^{31} theoretically possible execution paths

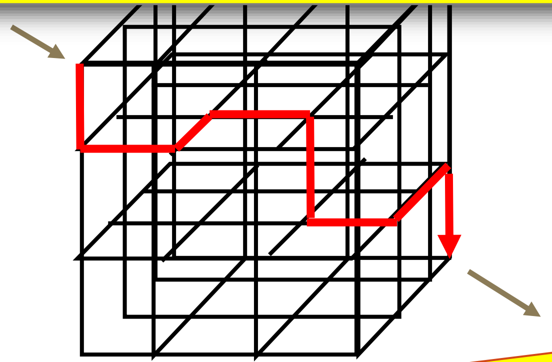
3: TESTING MULTI-THREADED CODE

So maybe MC/DC coverage is not such a great metric.
Can we do better with Fuzz Testing?

```
int    x
int   *p
int **a;

thread_1() // initialize
{
     p = &x;
     q = &y;
     z = &r;
}
```





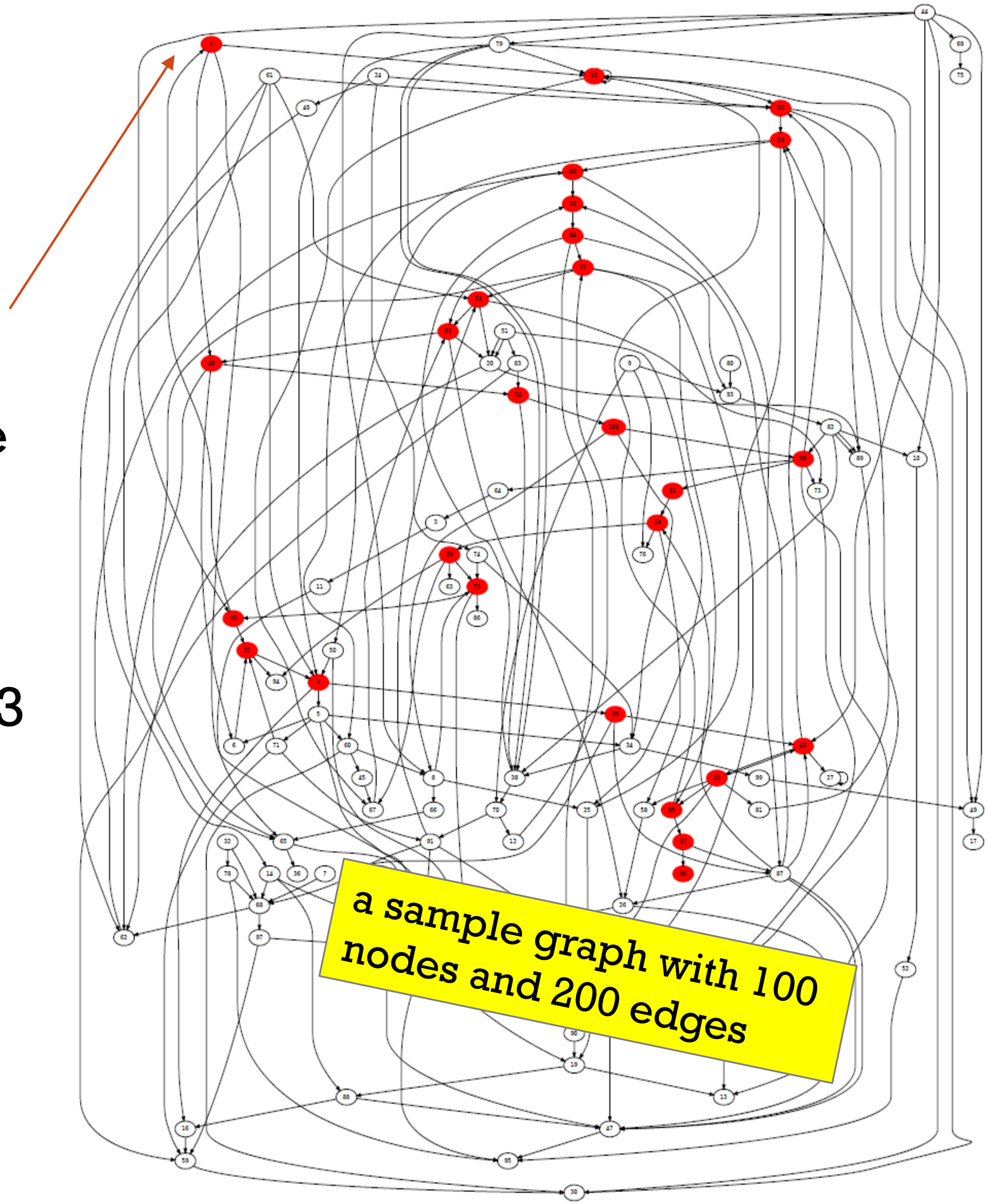
1,680 paths: but any ***1*** *of these reaches 100% MC/DC coverage*

```
thread_2() // swap *p and *q
{
     r = *p;
    *p = *q;
    *q = r;
}
```

```
thread_3() // acces                and p
{
       a = &p;
      *a = z;
     **a = 12;
}
```

AN EXAMPLE

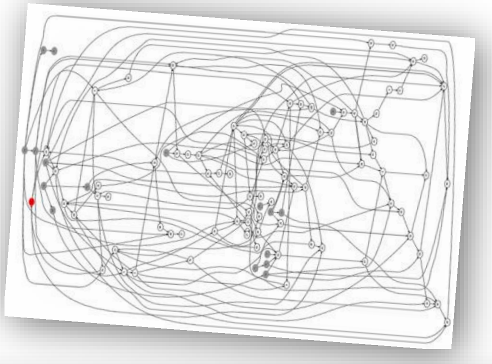
- 83 nodes are reachable from S1
- How many *random tests* would we have to do to be sure that all 83 nodes are visited at least once?
- Hint: a first randomly chosen test path shown here visits 27 of the 83 nodes, or 32.5% of the total.

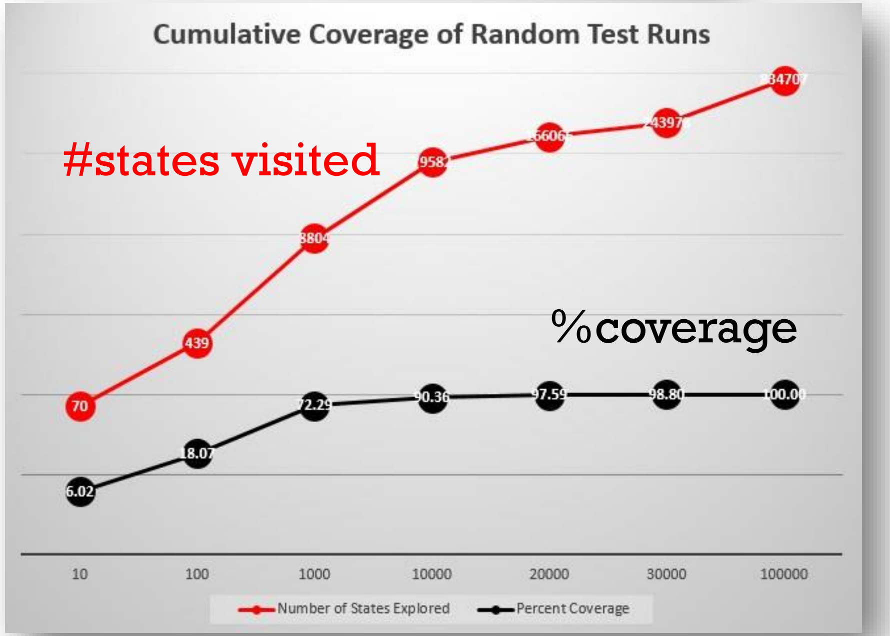


N RANDOM TESTS OF 500 STEPS

STATES VISITED VS UNIQUE STATES

nr of tests	visited states	unique states	percent coverage	runtime
10	70			1 second
100	439			3 seconds
1,000	8,804			1 minute
10,000	79,582			6 minutes
20,000	166,066			12 minutes
30,000	243,978			17 minutes
100,000	834,707			52 minutes

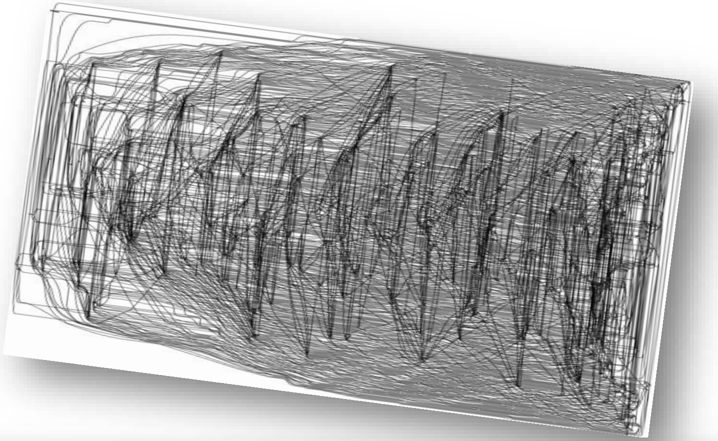




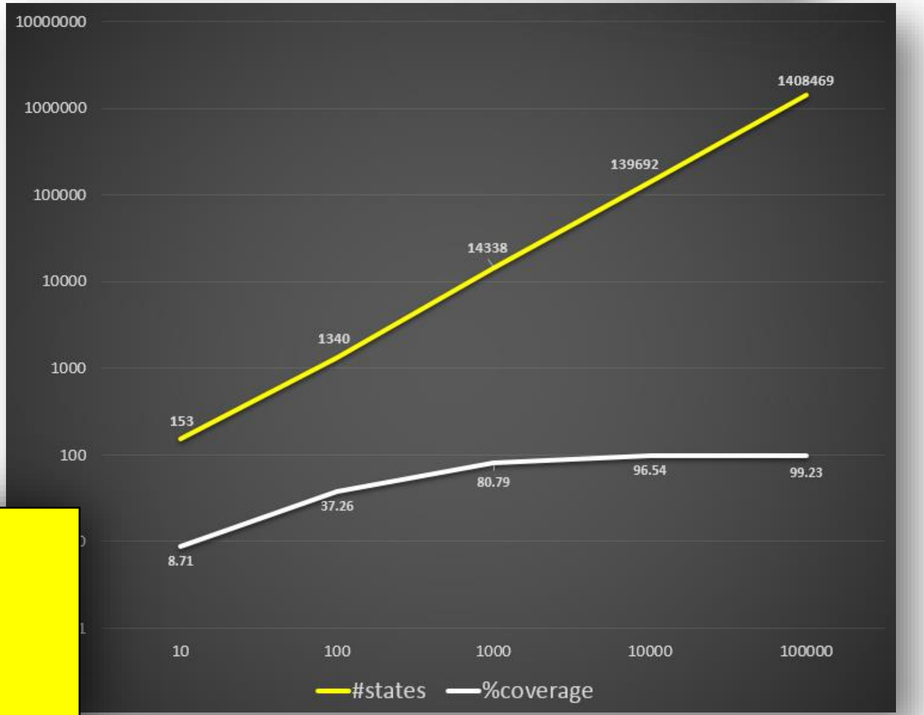
the x-axis (#tests) is a logscale

SAME TEST FOR A LARGER GRAPH

1000 NODES, 781 REACHABLE



nr of tests	visited states	unique states	percent coverage	time (sec)
10	153	68	9%	1
100	1,340	291	37%	6
1,000	14,338	631	81%	124
10,000	139,692	754	96%	640
100,000	1,408,469	775	99%	93120 (25.9 hrs)

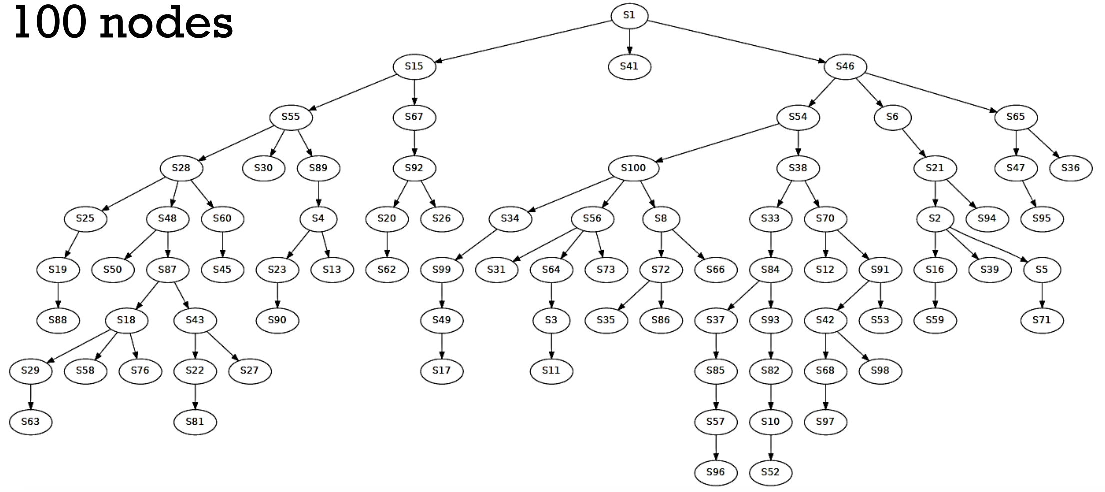


→ nr of random tests

so: random test suites are also not great:
they incur increasing amounts of ***duplicate*** work,
making it hard to reach 100% coverage

WHAT IF WE REMEMBERED WHERE WE'VE BEEN:
BY USING STANDARD GRAPH SEARCH ALGORITHMS (DFS/BFS)

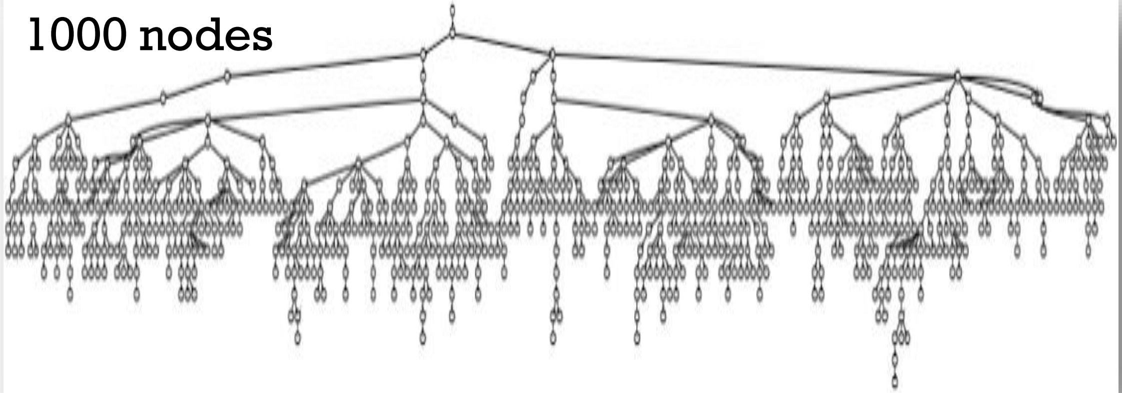
100 nodes



nr of tests	visited states	unique states	percent coverage	
1	83	83	100%	<1s

a standard breadth-first search (BFS) in either graph visits *all* reachable nodes and explores *all* execution paths, without duplication…
all in a fraction of a second

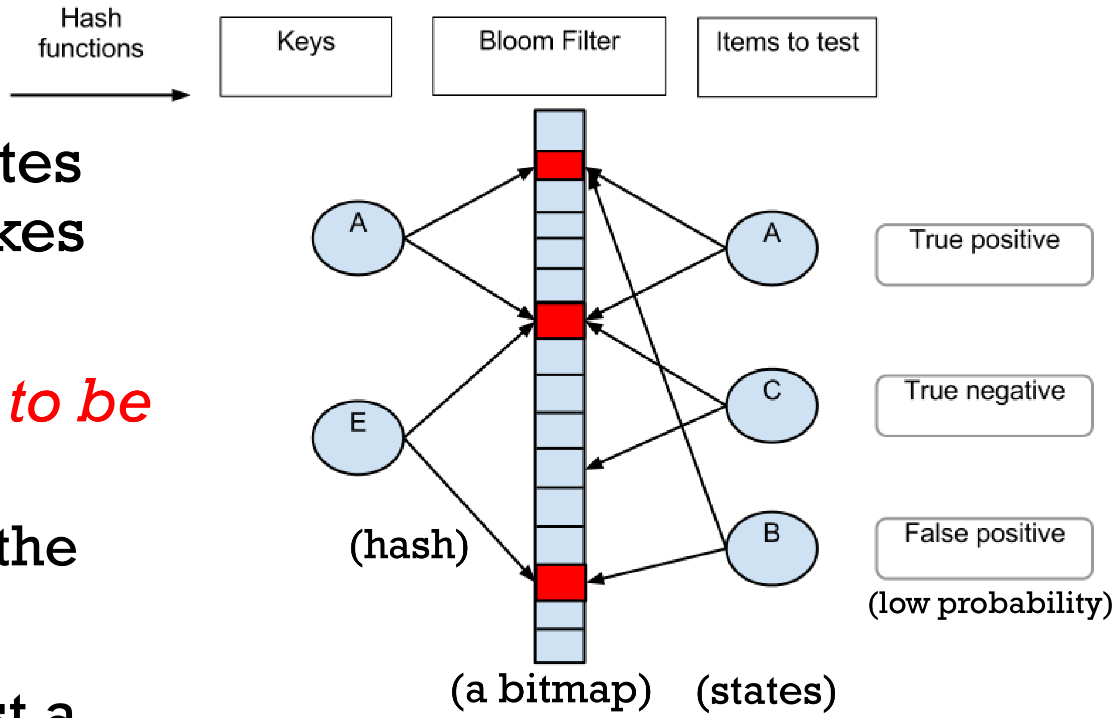
1000 nodes



nr of tests	visited states	unique states	percent coverage	
1	781	781	100%	<1s

THERE'S MORE…

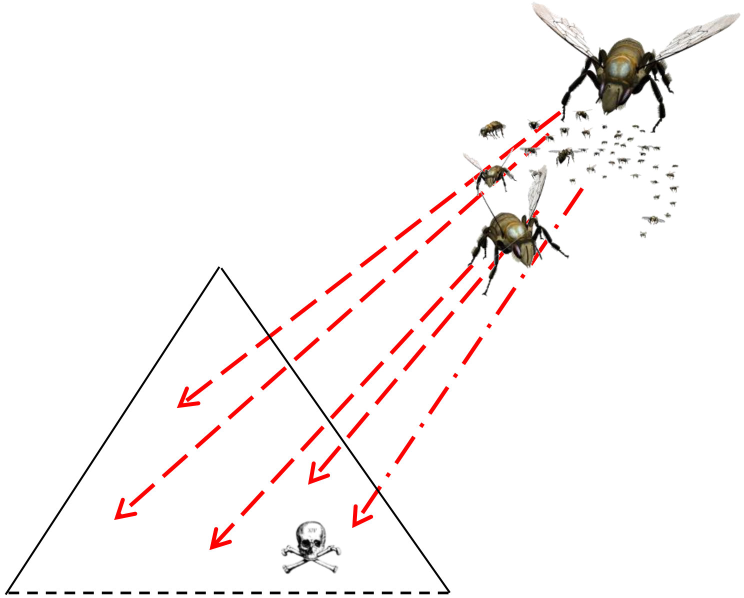
- What if storing all reachable states (for a perfect recall of states) takes too much memory?
- The good news: *it does not have to be perfect*
  - the recall is only used to *reduce* the amount of duplicate work
- It can already suffice to store just a hash-signature of each state
  - in a *fixed size* Bloom filter



Burton Bloom, "Space/time trade-offs in hash coding with allowable errors" CACM, July 1970, Vol. 13, Issue 7.

CAN WE EXPLOIT PARALLELISM TO CREATE VERY FAST BLOOM FILTER TESTS?

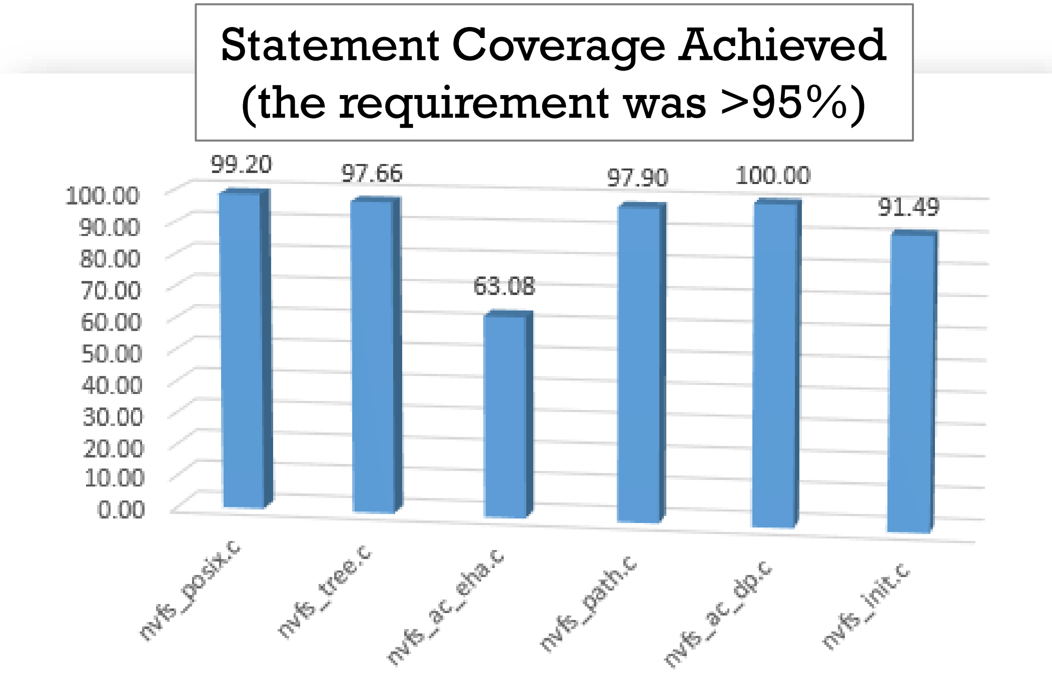
- for large problems, a full DFS or BFS search could be time consuming
- we can *parallelize* the tests if we randomly split up the search space: (re-enter *fuzzing* or *randomization*)
- *i've called this method: swarm testing*



method:

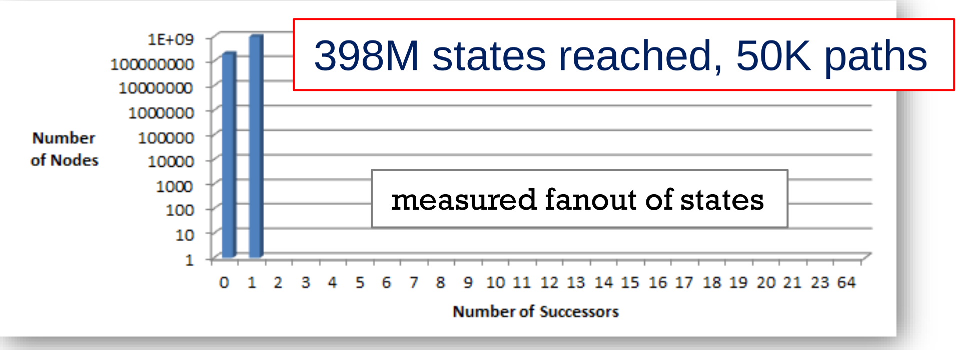
(1) N search engines (hundreds, thousands, millions)
(2) with a small memory bound for each search (fast!)
(3) randomize the DFS within each search engine
(4) achieves very high state coverage for large N

NVFS REQUIRED UNIT TESTS

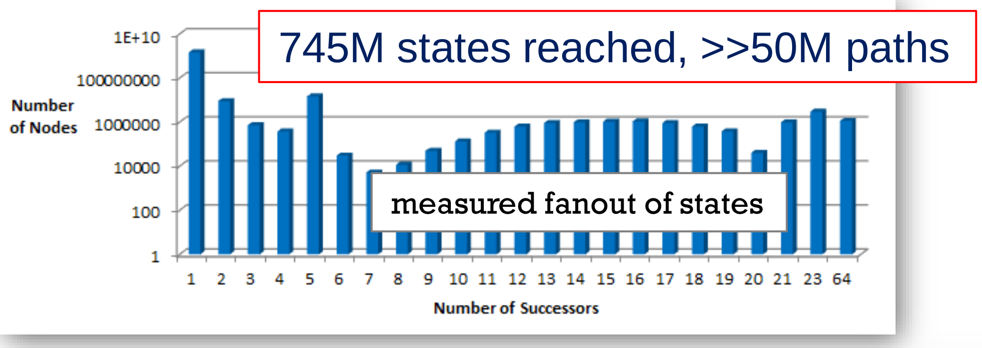


the number of unique system states
reached in all NVFS unit tests combined:
35,796 unique states (+ 1,175 duplicates)
and ~100 distinct test execution paths

After 5 hours of RANDOM TESTING



After 5 hours of BFS SEARCH (TWR)



The MC/DC Unit Tests explored
3 orders of magnitude fewer
states than either Random or BFS
BFS explored the largest number of paths

BUT NOTE: IT'S NOT JUST ABOUT EXPLORING ALL EXECUTION *PATHS*...

***10** execution paths*
(cyclomatic complexity 10)

```
int
function(int arg)
{       int result = 0;

        switch (p) {
        case 1: result = 5; break;
        case 2: result = 3; break;
        ….
        case 9: result = 2020; break;
        default:  break;
        }
        return result;
}
```

these two functions have
identical *functionality*

```
int table[10] = { 0, 5, 3, … , 2020 };

int
function(int arg)
{       int result = 0;

        if (arg >= 1 && arg <= 9)
        {       result = table[arg];
        }
        return result;
}
```

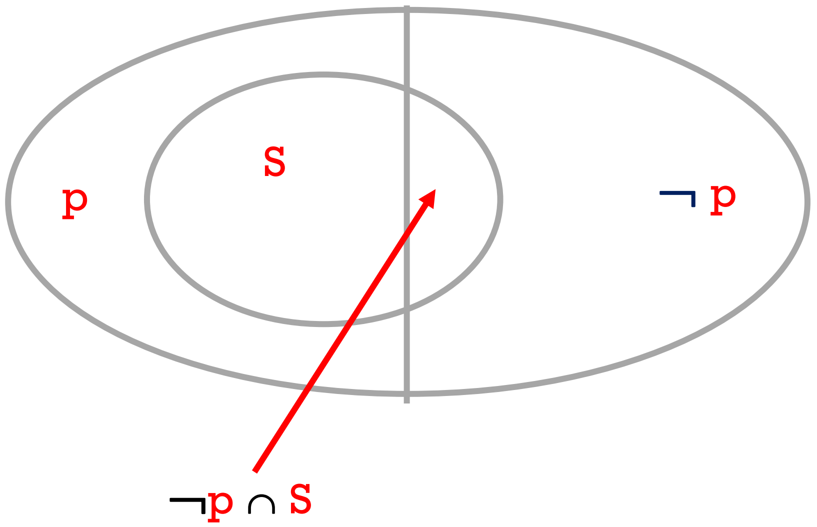
***2** execution paths*
(cyclomatic complexity 2)
an example of **data driven** *code*

FORMAL SOFTWARE ANALYSIS

given system S and a requirement p

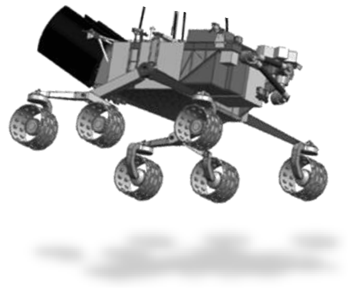
compute: $\neg p \cap S$

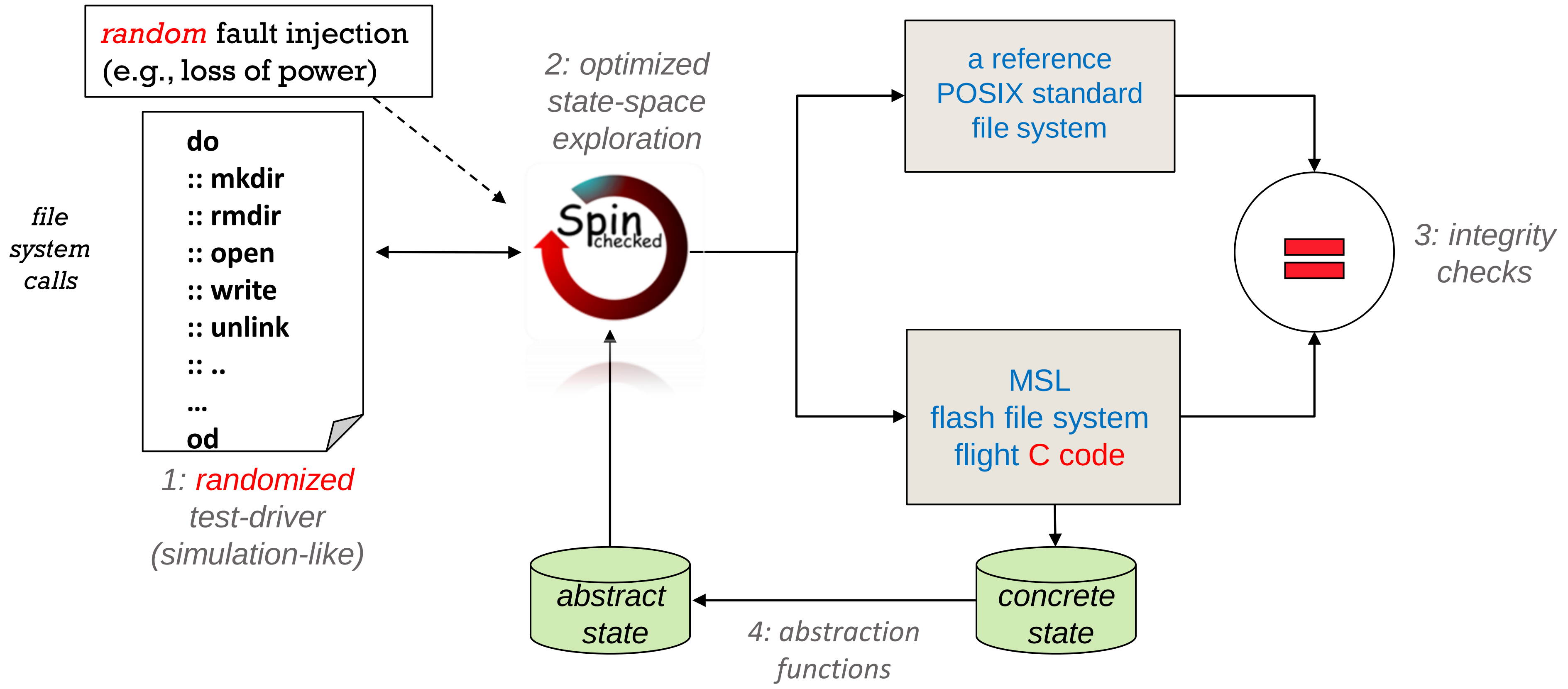
- p is expressed in (temporal) logic
- S captures (possibly concurrent) task behavior, using *partial order reduction* theory to reduce the search space



if the subset $\neg p \cap S$ is empty:
  we *prove* that p holds in S
if non-empty:
  the subset contains at least one execution that *proves* that p can be violated in S

HOW WE TESTED THE MSL ROVER's FLASH-FILE SYSTEM SOFTWARE





SYNOPSIS

- for *Testing with Recall*:
  - the application must be instrumented so that its *state* can be captured (hashed)

- by doing so we can:
  - *increase* test coverage (dramatically)
  - and perform *stronger* checks:
  - use full linear temporal logic model checking
  - use cloud computing techniques to speed up the testing

THANK YOU !

"A random element is rather useful when we are searching for a solution of some problem.“

A.M. Turing, "Computing machinery and intelligence," Oxford University Press, MIND (the Journal of the Mind Association), Vol. LIX, no. 236, pp. 433-60, (**1950**).