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 the outcome.

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?
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

```c
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
So maybe MC/DC coverage is not such a great metric. Can we do better with Fuzz Testing?
**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.
# RANDOM TESTS OF 500 STEPS

## STATES VISITED VS UNIQUE STATES

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

the x-axis (#tests) is a logscale
SAME TEST FOR A LARGER GRAPH

1000 NODES, 781 REACHABLE

<table>
<thead>
<tr>
<th>nr of tests</th>
<th>visited states</th>
<th>unique states</th>
<th>percent coverage</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>153</td>
<td>68</td>
<td>9%</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1,340</td>
<td>291</td>
<td>37%</td>
<td>6</td>
</tr>
<tr>
<td>1,000</td>
<td>14,338</td>
<td>631</td>
<td>81%</td>
<td>124</td>
</tr>
<tr>
<td>10,000</td>
<td>139,692</td>
<td>754</td>
<td>96%</td>
<td>640</td>
</tr>
<tr>
<td>100,000</td>
<td>1,408,469</td>
<td>775</td>
<td>99%</td>
<td>93120</td>
</tr>
</tbody>
</table>

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)

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

<table>
<thead>
<tr>
<th>nr of tests</th>
<th>visited states</th>
<th>unique states</th>
<th>percent coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>83</td>
<td>83</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;1s</td>
</tr>
</tbody>
</table>

100 nodes

<table>
<thead>
<tr>
<th>nr of tests</th>
<th>visited states</th>
<th>unique states</th>
<th>percent coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>781</td>
<td>781</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;1s</td>
</tr>
</tbody>
</table>

1000 nodes
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”
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 $N$
The MC/DC Unit Tests explored 3 orders of magnitude fewer states than either Random or BFS. BFS explored the largest number of paths.

NVFS REQUIRED UNIT TESTS

Statement Coverage Achieved (the requirement was >95%)

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
398M states reached, 50K paths
measured fanout of states

After 5 hours of BFS SEARCH (TWR)
745M states reached, >>50M paths
measured fanout of states

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;
}

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)

these two functions have identical functionality

an example of data driven code
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

random fault injection
(e.g., loss of power)

do :: mkdir :: rmdir :: open :: write :: unlink :: ...

do

1: randomized test-driver (simulation-like)

abstract state

2: optimized state-space exploration

a reference POSIX standard file system

MSL flash file system flight C code

concrete state

3: integrity checks

4: abstraction functions
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
"A random element is rather useful when we are searching for a solution of some problem."