WHAT’S THE FUZZ ABOUT TESTING?

 вопросы

1. Как хорошо Unit Testing с
100% MC/DC Coverage?
2. Использует (Randomized
Testing) лучше?
3. Что, если мы используем Testing with
Perfect Recall?
4. Как мы можем использовать Parallelism?

“Whatever can happen will happen
if we make trials enough.”
Augustus De Morgan (1866)
THREE SMALL EXAMPLES
1: CONDITIONALS

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

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

This test achieves 100% MC/DC coverage, yet it misses the bug that is revealed with another test: `foo(0,1)`

It covers just 50% of the execution paths in the control-flow graph.

2: LOOPS

```c
void fct(int x, int y)
{
    int i, a[4];
    for (i = 0; i < x+y; i++)
    {
        a[i] = i;
    }
}
```

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

This test achieves 100% MC/DC coverage, yet it misses the array indexing bug that is revealed with, for instance, `foo(1,3)`

It covers just 1 of $2^{31}$ theoretically possible execution paths.
3: THREADS

```c
int x, y, r;
int *p, *q, *z;
int **a;

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

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

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

So maybe MC/DC coverage is not the best metric to aim for. Is Random Test Selection (Fuzz Testing) better?

AN EXAMPLE

- 83 nodes are reachable from the node labeled S1
- How many random tests would we have to do to be sure that all 83 reachable nodes are visited at least once?

a sample graph with 100 nodes and 200 edges
### N 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>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>70</td>
<td>5</td>
<td>6%</td>
<td>1 second</td>
</tr>
<tr>
<td>100</td>
<td>439</td>
<td>15</td>
<td>18%</td>
<td>3 seconds</td>
</tr>
<tr>
<td>1,000</td>
<td>8,804</td>
<td>60</td>
<td>72%</td>
<td>1 minute</td>
</tr>
<tr>
<td>10,000</td>
<td>79,582</td>
<td>75</td>
<td>90%</td>
<td>6 minutes</td>
</tr>
<tr>
<td>20,000</td>
<td>166,066</td>
<td>81</td>
<td>97%</td>
<td>12 minutes</td>
</tr>
<tr>
<td>30,000</td>
<td>243,978</td>
<td>82</td>
<td>99%</td>
<td>17 minutes</td>
</tr>
<tr>
<td>100,000</td>
<td>834,707</td>
<td>83</td>
<td>100%</td>
<td>52 minutes</td>
</tr>
</tbody>
</table>

(the x-axis is a logscale)

### THE SAME TEST FOR A GRAPH OF 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>

random test is unbiased, but does increasing amounts of duplicate work as it progresses, making it hard to push coverage to 100%
WHAT IF YOU COULD REMEMBER WHERE YOU’VE BEEN: WITH DFS OR BFS

![Diagram of graphs with BFS](image)

<table>
<thead>
<tr>
<th></th>
<th>nr of tests</th>
<th>visited states</th>
<th>unique states</th>
<th>percent coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 nodes</td>
<td>1</td>
<td>83</td>
<td>83</td>
<td>100%</td>
</tr>
<tr>
<td>1000 nodes</td>
<td>1</td>
<td>781</td>
<td>781</td>
<td>100%</td>
</tr>
</tbody>
</table>

A breadth-first search (BFS) with Perfect Recall in either graph visits all reachable nodes and explores all execution paths without duplication in seconds.

DOES THE RECALL HAVE TO BE PERFECT?

- What if storing all reachable states (for perfect recall) takes too much memory?
- That’s Okay: Recall does not have to be completely Perfect: it is only meant to reduce the amount of duplicate work
- It often suffices to store just a hash-signature (or just a few bits using a fixed size Bloom filter)
CAN IT BE FAST TOO?

- For large problems, a full DFS or BFS search could require excessive amounts of time.
- But, we can parallelize the tests if we can split up the search space, using ... randomization leading to a technique for Swarm Testing.

method:
(1) use N parallel search engines (hundreds, thousands)
(2) define a small memory bound M for each search
(3) randomize the DFS within each search engine

AN EXPERIMENT

NVFS UNIT TEST SUITE

The arget Statement Coverage (%) for MSL was 95-100%

After 5 hours of RANDOM TESTING

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

After 5 hours of BFS SEARCH (TWR)

- 745M states, >>50M paths
- measured fanout of states

Despite ~98% statement coverage, the Unit Tests explored 3 orders of magnitude fewer states than either Random or TWR. Testing with Recall (TWR) performed best.

We measured the number of unique system states reached in all the above NVFS unit tests combined as 35,796 unique states (plus 1,175 duplicates) and an estimated number of 100 distinct execution paths.
SO WHAT IS THE FUZZ?

- There's one downside: to use *Testing with Recall* the application must be instrumented so its *state* can be captured and remembered.
- But if you do this you can:
  - dramatically increase test coverage
  - and, you can also perform much stronger checks, up to full linear temporal logic model checking of source code.

THANK YOU

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

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.