

Quantitative Sampling for Probabilistic Assert Failures

Sumit Lahiri
IIT Kanpur

March 18, 2021

We try to formulate a way to compute path probabilities using symbolic execution and testing based technique.

```
int main(void)
{
    int a; // uninitialized
    int d = std::uniform_distribution<rd_seed>(0, 650);

    // forall variable : (INT_MIN to INT_MAX)
    klee_make_symbolic(&a, sizeof(a), "a_sym");

    // PSE variable : Uniformly distributed [0 to 650]
    make_pse_symbolic<int>(&d, sizeof(d), "d_prob_sym", 0, 650);

    int c = a + 100;

    // case 1 : Pure Forall Predicate
    if (a > 50) {
        c = a + 75;
    } else {
        c = a - 75;
    }

    // case 2 : Pure PSE Predicate
    if (d > 60) d = 250;

    // case 3 : Dependence Case
    if (c > d) c = d;

    // Probabilistic query : assert(P(c != d) < 0.5)
    // Optimize here :
    //     Optimal value of forall variable 'a'
```

```

    //      such that  $P(c \neq d)$  is close to 0.5
    return 0;
}

```

Algorithm 1 Candidates : (Testing Based Estimation)

```

1: for each  $p \in Paths$  do
2:    $c := ConstraintSet(p)$  ▷ Path Constraints for p
3:    $m := Optimize(query, c)$  ▷ solution for the path constraints
4:    $concreteSet = \{\}$ 
5:   for each  $v \in ForallVars(p)$  do ▷ ForallVars p → forall
6:      $concreteSet.append(\{key : v, val : m[v]\})$  ▷ Candidate Values
7:   end for each
8:    $executeCV(program, concreteSet)$ 
9: end for each

```

Algorithm 2 executeCV : PSE Sampled Normal Execution

```

1: function EXECUTECV( $P : program, C : concreteSet$ )
2:   for each  $v \in ForallVars(p)$  do
3:      $value(v) := concreteSet(v)$  ▷ Use values from ConcreteSet
4:   end for each
5:   ... ▷ proceed with normal execution
6: end function

```

For the sample program given above, we first resort to using **symbolic execution** to generate **path constraints** for all the feasible paths that this program can take and then convert the **path constraints** into an **formal logic optimization problem** that gives an **assignment** to **forall** variables such that it leads to optimum violation of the *query*.

```

def generateCandidates(k: int): # Candidates Algorithm
    opt = z3.Optimize()
    a = z3.Int("a_sym")
    d = z3.Int("d_prob_sym")

    opt.add(d >= 0)
    opt.add(d <= 650)
    opt.add(a > 50)
    opt.add(z3.Not(d > 60))
    opt.add(a + 75 > d)

    opt.maximize(a - d - 75) # Query to optimize
    n = 0
    while opt.check() == z3.sat and n != k:

```

```

m = opt.model()
n += 1
print("%s = %s" % (a, m[a]))
print("%s = %s" % (d, m[d]))
opt.add(a != m[a])

```

We now explore a slightly different example which is more involved in terms of the constraints and query that the user can pose at the end of the symbolic execution. In the below example we bound the values for `forall`s for example sake.

```

// forall variable
klee_make_symbolic(&a, sizeof(a), "a_sym");           // [0, 1]
klee_make_symbolic(&c, sizeof(b), "c_sym");           // [1, 10]
klee_make_symbolic(&d, sizeof(c), "d_sym");           // [0, 5]
klee_make_symbolic(&win, sizeof(win), "win_sym");     // win == 1

// PSE variable
make_pse_symbolic<int>(&b, sizeof(b), "b_prob_sym", 0, 1);
make_pse_symbolic<int>(&e, sizeof(e), "e_prob_sym", 1, 6);

klee_assume(a >= 0 && a <= 1);
klee_assume(c >= 1 && c <= 10);
klee_assume(d >= 0 && d <= 5);

if (a > b)                                           // maximize a
{
    if (c + e < 15)                                 // minimize c
    {
        win = 1;
        win_ones++;
    }
    else
    {
        win = 0;
        win_zeros++;
    }
}
else
{
    if (d + e > 1)                                   // maximize d
    {
        win = 1;
        win_ones++;
    }
    else

```

```

{
    win = 0;
    win_zeros++;
}
}

```

Below we show a sample of the type of the **queries** that a user can make in the context of the example shown above.

```
assert(P(win == 1) > 0.8);
```

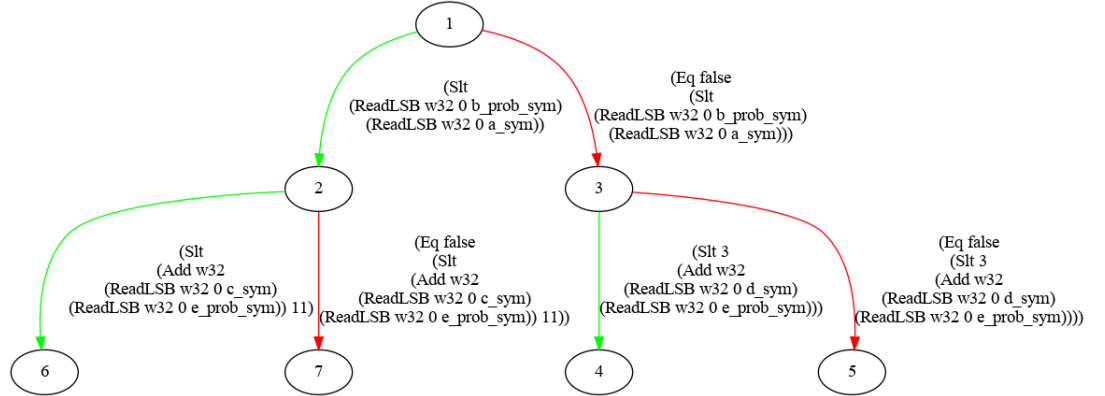
Based on the query posed, we get **candidate** models by converting the code into a **optimization** query and solve this optimization problem using any off-the-shelf **SMT Solver** to get model values for the forall variables that contribute to maximum violation of the **query** constraint.

Based on the example we do the following **optimizations**. one on Path 1 and the other on Path 3 where `win == 1`

```
Path 1 : maximize((a - b) + 11 - (c + e))
```

```
Path 3 : maximize((b - a) + (d + e) - 1)
```

On the other two paths, we **minimize** the same. `win == 1` on the **green** edged paths.



```

Path 1 : [And(a_sym >= 0, a_sym <= 1), And(c_sym >= 1, c_sym <= 10),
And(d_sym >= 0, d_sym <= 5), And(b_prob_sym >= 0, b_prob_sym <= 1),
And(e_prob_sym >= 1, e_prob_sym <= 6), win_sym == 1, b_prob_sym <
a_sym, c_sym + e_prob_sym < 15]

```

```

Model : 1
    a_sym = 1
    c_sym = 1
    d_sym = 0

```

```
Model : 2
```

```

        a_sym = 1
        c_sym = 2
        d_sym = 0
    Model : 3
        a_sym = 1
        c_sym = 3
        d_sym = 0
Path 2 : [And(a_sym >= 0, a_sym <= 1), And(c_sym >= 1, c_sym <= 10),
And(d_sym >= 0, d_sym <= 5), And(b_prob_sym >= 0, b_prob_sym <= 1),
And(e_prob_sym >= 1, e_prob_sym <= 6), b_prob_sym < a_sym, Not(c_sym
+ e_prob_sym < 15)]
    Model : 1
        a_sym = 1
        c_sym = 9
        d_sym = 0
    Model : 2
        a_sym = 1
        c_sym = 10
        d_sym = 0
Path 3 : [And(a_sym >= 0, a_sym <= 1), And(c_sym >= 1, c_sym <= 10),
And(d_sym >= 0, d_sym <= 5), And(b_prob_sym >= 0, b_prob_sym <= 1),
And(e_prob_sym >= 1, e_prob_sym <= 6), win_sym == 1, Not(b_prob_sym
< a_sym), d_sym + e_prob_sym > 1]
    Model : 1
        a_sym = 0
        c_sym = 1
        d_sym = 0
    Model : 2
        a_sym = 0
        c_sym = 2
        d_sym = 0
    Model : 3
        a_sym = 0
        c_sym = 3
        d_sym = 1

    ...

```

Here $P(\text{win} == 1)$ is the query of interest to us so we optimize along those paths where this condition holds.

We now do a **transformation** pass over the program and instrument the count of the given **condition** failing. In the **transforamtion** pass, we make the **program** take values that we find as **candidate** models upon following Algorithm 1

```
// Take in candidate vector for [foralls]
```

```

scanf("%d", &a); // [0, 1]
scanf("%d", &c); // [1, 10]
scanf("%d", &d); // [0, 5]

// // PSE variable : Random Sampling
std::default_random_engine generator;
std::uniform_int_distribution<int> distribution1(0, 1); // b
std::uniform_int_distribution<int> distribution2(1, 6); // e

while (term_count--> 0) // term_count > 500000 (large sample)
{
    b = distribution1(generator);
    e = distribution2(generator);

    if (a > b) { //
        if (c + e < 15) {
            win = 1;
            win_ones++;
        } else {
            ...
        }
    } else {
        if (d + e > 1) {
            win = 1;
            win_ones++;
        } else {
            ...
        }
    }
    run++;
}

// The probabilistic assert is the following
// assert(P(win == 1) >= 0.8), where P(win == 1) win_ones/run;

```

We tweaked the values in such a way that for a very small number of candidate vectors the (probabilistic) assert fails and our algorithm must now catch that. Indeed we find that for the following assignments to **forall** variables, the assert fails.

```

Fail : P(win == 1) : 0.754100
Vals -> a : 1, c : 10, d : 0

```

For other **candidate** vectors, we find that the **probabilistic** asserts actually hold and will not cause a violation during execution under normal conditions. We show the results of some of the winning cases here.

```

Pass : P(win == 1) : 1.000000
      Vals -> a : 0, c : 7, d : 4
Pass : P(win == 1) : 1.000000
      Vals -> a : 1, c : 4, d : 4
Pass : P(win == 1) : 1.000000
      Vals -> a : 1, c : 1, d : 3
Pass : P(win == 1) : 1.000000
      Vals -> a : 0, c : 6, d : 4
Pass : P(win == 1) : 0.834900
      Vals -> a : 0, c : 7, d : 0
Pass : P(win == 1) : 0.834900
      Vals -> a : 0, c : 2, d : 0
Pass : P(win == 1) : 1.000000
      Vals -> a : 0, c : 10, d : 2
Pass : P(win == 1) : 0.917800
      Vals -> a : 1, c : 3, d : 0
      ...

```