PSE Quant Sampling Algorithm

Sumit Lahiri

February 26, 2021

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

```
int main(void)
{
        int a; // unintialized
        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 (a) such that P(c != d) is close to 0.5
        return 0;
}
```

Algorithm 1 Dependence Case : (Testing Based Estimation)

```
for each p ∈ Paths do
    c := ConstraintSet(p)                          ▷ Path Constraints for p
    m := Optimize(query, c)                        ▷ solution for the path constraints
    concreteSet = {}
    for each v ∈ ForallVars(p) do                  ▷ ForallVars p → forall
        concreteSet.append({key : v, val : m[v]})  ▷ Candidate Values
    end for each
    executeCV(program, concreteSet)
end for each
```

Algorithm 2 executeCV : PSE Sampled Normal Execution

```
function EXECUTECV(P : program, C : concreteSet)
    for each v ∈ ForallVars(p) do
        value(v) := concreteSet(v)                 ▷ Use values from ConcreteSet
    end for each
    ...                                            ▷ proceed with normal execution
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*.

```python
def generateCandidates(k: int):
        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])
```