Week-7

L1 Symbolic Testing-1

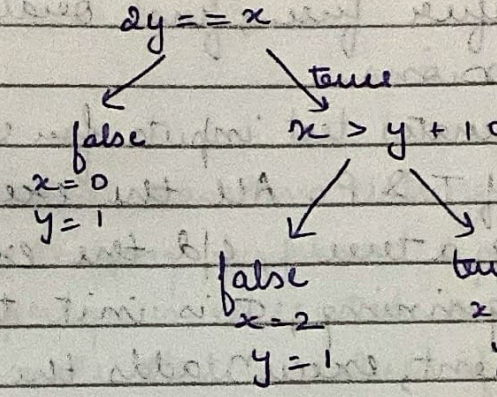
- Symbolic execn is a means of analyzing a given prog; to determine what inputs cause each prog. part to execute. It is used to exercise diff. execn's of the prog. Reachability & infecn problems for logic-based testing can be solved by sym. execn.

-
```
1 sum(a,b,c)
2   x = a+b
3   y = b+c
4   z = x+y-b
5   return z
  end.
```

Normal						
	x	y	z	a	b	c
1.	–	–	–	1	3	5
2.	4	–	–	–	–	–
3.	8	–	–	–	–	–
4.	–	–	9	–	–	–
5. return 9						

Symbolic. execn.					
x	y	z	a	b	c
1. –	–	–	α_1	α_2	α_3
2. $\alpha_1+\alpha_2$	–	–	–	–	–
3. –	$\alpha_2+\alpha_3$	–	–	–	–
4. –	–	$\alpha_1+\alpha_2+\alpha_3$	–	–	–
5. returns $(\alpha_1+\alpha_2+\alpha_3)$					

-
```
x = input()
y = input()
int twice(int v){
  return 2v
}
main(int x, y){
  z = twice(y)
  if z == x
    if (x > y+10)
      then
```



L2 Symbolic Testing-2

- Program proving involves proving programs correct. It is diff. from testing, where a prog. is tested only against a set of test cases.
- Formal methods → involves mathematical techniques that are based on deriving formal methods from a prog. or sys. under test. They have mathematically sound semantics. 3 techniques → model checking, theorem proving, prog. analysis.
- In testing, one can be assured that sample test runs work correctly by carefully checking the results.

Correct execu^n of prog. for inputs not in the sample is still in doubt.

- In proving, we formally prove that prog. meets its specifica^n w/o executing the prog. at all. They might need to done by hand.
- Symbolic execu^n → testing technique that executes a prog. symbolically for a set of inputs. It uses symbolic values instead of concrete data values. Prog. variables are represented as symbolic exp. over the input values. Symbolic state σ maps vars. To sym. expressions. Sym. PC (path constraint) is a quantifier free, first order formula over sym. expressions.
- It generates test inputs for each execu^n path. It is a seq. of T & F. All the execu^n paths are represented using a tree, c/d the execu^n tree.
- At beginning, σ is init to empty map. At the read statement, execu^n adds the mapping assigning the var. being read to σ. At assignment $v = e$, update σ by mapping $v \to \sigma(e)$
- Prev. eg. $\sigma = \{x \mapsto x_0, y \mapsto y_0\}$ & so on
- At the end of symbolic execu^n, PC is solved using a constraint solver to generate concrete input values. It can also be terminated if the prog. hits an exit statement or error.

Eg.

```
sum = 0
N = sym.input()
while N>0:
    sum = sum + N
    N = sym.input()
```

PC → $\left(\bigwedge_{i \in [1,n]} N_i > 0\right) \wedge (N_{n+1} \leq 0)$

σ → $\{N \mapsto N_{n+1}, sum \mapsto \sum_{i \in [1,n]} N_i\}$

- In gen, symb. execun of code containing loops may result in an ∞ no. of paths if the Terminan condition for the loop is symbolic.
- Symb. testing is dep. on constraint solver that takes a path constraint & gives a set of input values to be used as test cases. If PC can't be efficiently solved by a constraint solver, we can't generate test inputs. ⇒ Key drawback.
- Modern day SMT solvers can handle much amt. of data. Comp. have more memory & hence, process data faster. ⇒ Concolic Testing

L3 Concolic Testing - I

- DART (directed automated random testing) or concolic (concrete + symbolic) testing performs sym. execun dynamically, while prog. is executed on some concrete inputs. Concrete state: values of all vars in the prog.
- We gen. random input for the prog. Collect symbolic PC along this execun. Gen. inputs that derives prog. along diff. execun paths. Repeat till all execun paths are explored.
- DART, three main techniques —
  1. automated extract^n of the interface of a prog. with its ext. envt. using static source code parsing.
  2. automatic genera^n of test drivers for this interface that performs random testing to simulate the most gen. envt. the prog. can operate in.
  3. Dynamic analysis of how prog. behaves under random testing & automatic genera^n of new tests

inputs to direct systematically the execn along alternative prog. paths.
- DART dynamically gathers knowledge abt. the execn of prog., also c/d directed search. PC is a vector that contains values that are soln of symb. constraints gathered from predicates in branch statements during prev. execn.

L7.4 Concolic execn - 2
- Execn model of dart helps to execute the prog. normally & symbolically. In symbolic execn, DART collect PC over symbolic var. & solves the constraints to gen. values for next test case.
- Memory (M) ⇒ a mapping from memory addresses m to say 32-bit words. + ⇒ updating memory.
- Symbolic var. ⇒ identified by their addresses.
- Symb. exp. ⇒ exp.e ⇒ set of addresses m that occur in it.
- abort (prog. error) or halt (prog. execn finished)
- DART Th. →
1. if run-DART prints "Bug Found" ⇒ some input of P leads to an abort.
2. if vice-versa to 1 ⇒ all paths in Execs(P) have been exercised.
3. run-DART runs forever.

LS Summary

- DART distinguish 3 types of funcs –
  1. prog. func^n → defined in the prog.
  2. ext. func^n → func^{ns} controlled by envt. & hence part of ext. interface.
  3. library func^n → not defined in prog., but are controlled by the prog.
- DART uses constraint solver IP_solve for efficiently solving any linear constraint using real & int. programming techniques.