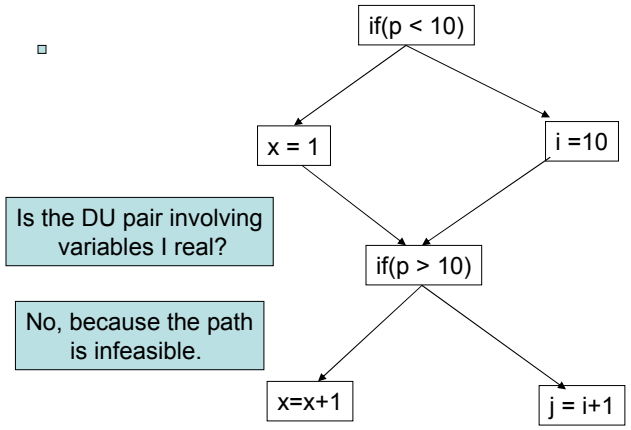
Symbolic Execution

Saswat Anand
22/09/2009

Limitation of Dataflow Analysis



Outline

- Background
  - feasible and infeasible program paths
  - constraints, and constraint satisfiability
- Symbolic execution
  - base idea
  - handling of symbolic references
- Overview of compositional symbolic execution
- Overview of implementation of symbolic execution
- Limitations of symbolic execution
- Summary

Feasible and Infeasible Paths

- A *path* refers to a path in the (inter-procedural) control-flow graph of the program.
- A path is *feasible* if there exists an input I to the program that covers the path; i.e., when program is executed with I as input, the path is taken.
- A path is *infeasible* if there exists no input I that covers the path.

Infeasible Paths

- Infeasible path does not imply dead code; However dead code implies infeasible path.
- In all real software, a very large portion of the total no. of paths are infeasible.
- Automatic test-input generation does not scale when there are large no. of infeasible paths to the target location that needs to be covered.

```
If(sameGoto)
   newTarget =
       ((IfStmt) stmtSeq[5]).getTarget();
else {
   newTarget = next;
   oldTarget =
       ((IfStmt) stmtSeq[5]).getTarget();
}
…
If(!sameGoto)
   b.getUnits().insertAfter(…);
…
```

An example of infeasible path from soot. A path that goes through the then branches of both conditional stmt.s is infeasible.

Constraints

$$X > Y \wedge Y+X \leq 10$$

- X, Y are called ***free variables***.
- A ***solution*** of the constraint is a set of assignments, one for each free variable that makes the constraint satisfiable.
- {X = 3, Y=2} is a solution.
- {X = 6, Y=5} is not a solution.

More types of constraints

1. Linear constraint
   - $X > Y \wedge Y+X \leq 10$
2. Non-linear constraint
   - $X * Y < 100$
   - $X \% 3 \wedge Y > 10$
   - $(X >> 3) < Y$
3. Use of function symbols
   - $f(X) > 10 \wedge (\text{forall } a.\ f(a) = a + 10)$

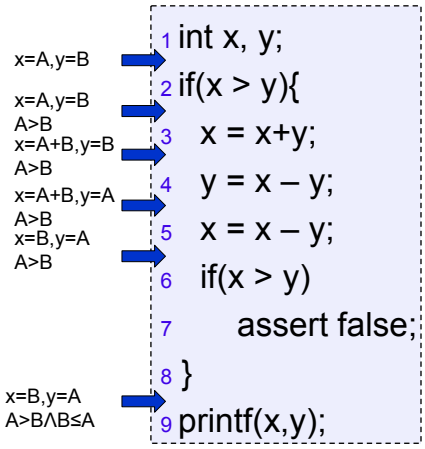
Constraints (contd.)

- A ***decision procedure*** is a tool that can decide if a constraint is satisfiable.
- A ***constraint solver*** is a tool that finds satisfying assignments for a constraint, if it is satisfiable.
- In general, checking constraint satisfiability is undecidable.

Symbolic Execution

- Symbolic execution refers to execution of program with symbols as argument.
- Unlike concrete execution, where the taken path is determined by the input, in symbolic execution the program can take ***any*** feasible path.
- During symbolic execution, program state consists of
  - symbolic values for some memory locations
  - path condition
- Path condition is a condition on the input symbols such that if a path is feasible its path-condition is satisfiable.
- Solution of path-condition is an test-input that covers the respective path.

Symbolic Execution

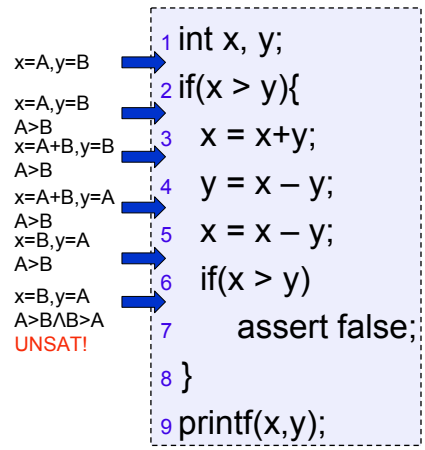


inputs that cover ***else*** branch at stmt. 2:
x = **3** y = **4**

inputs that cover ***then*** branch at 2 and ***else*** at 6:
x = **5** y = **1**

One solution of the constraint A>B Λ B≤A is
A = 5, B = 1

Symbolic Execution

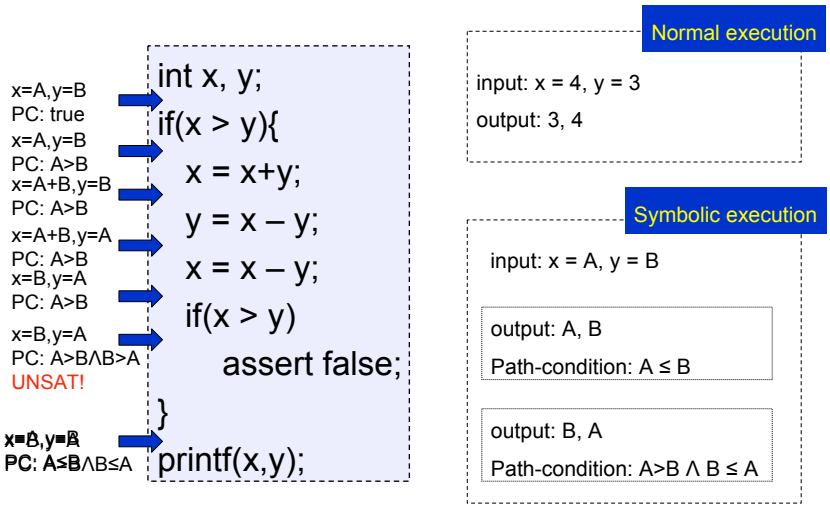


inputs that cover ***else*** branch at stmt. 2:
x = **3** y = **4**

inputs that cover ***then*** branch at 2 and ***else*** at 6:
x = **5** y = **1**

inputs that cover ***then*** branch at 2 and ***then*** at 6:
Does not exist!

All-paths Symbolic Execution



Handling Symbolic References

```
1 class Node {
2   int elem;
3   Node next;

4   foo(Node n1, Node n2){
5     if(n1 == null)  return;
6     if(n2 == null)  return;
7     if (n2.elem == 0)
8       return;
9     if (n1.next != null)
10      n1.next.elem = n1.elem -10;
11    assert(n2.elem != 0);
12  }
```

Handling Symbolic References

- setElem(H,n,e) – updates the elem field of node n in heap H to value e; returns the updated heap
- getElem(H,n) – returns the value of elem field of node n in heap H
- setNext(H,n,e), getNext(H,n) – likewise for next field

Invariants:

forall H, n. getElem(setElem(H,n,v),n) = v

forall H, n. getNext(setNext(H,n,v),n) = v

Handling Symbolic References

```
1 class Node {
2   int elem;
3   Node next;

4   foo(Node n1, Node n2){
5     if(n1 == null)  return;
6     if(n2 == null)  return;
7     if (n2.elem == 0)
8       return;
9     if (n1.next != null)
10      n1.next.elem = n1.elem -10;
11    assert(n2.elem != 0);
12  }
```

Path condition for the path 4-5-6-7-9-10-11

$n1 \neq null \wedge$
$n2 \neq null \wedge$
$getElem(H1,n2) \neq 0 \wedge$
$getNext(H1,n1) \neq null \wedge$
$H2 = setElem(H1,$
$\quad getNext(H1,n1),$
$\quad getElem(H1,n1)-10) \wedge$
$getElem(H2,n2) = 0$

Compositional Symbolic Execution

```
int abs(int x){
  if(x >= 0)
    return x;
  else
    return –x;
}

int sumAbs(int[] a){
   int sum = 0;
   for(int i = 0; i < 50; i++)
      sum += abs(a[i]);
   if(sum == 13)
      error();
   return sum;
}
```

- Goal: generate an input that covers leads to execution of error()
- No. of paths to error() = 2^{50}
- Symbolically executing each path and checking its feasibility does not scale!
- Key idea: compute function summaries to be used at all call-sites of the function

Compositional Symbolic Execution (contd.)

```
int abs(int x){
  if(x >= 0)
    return x;
  else
    return –x;
}

int sumAbs(int[] a){
   int sum = 0;
   for(int i = 0; i < 50; i++)
      sum += abs(a[i]);
   if(sum == 13)
      error();
   return sum;
}
```

- Symbolically execute all paths of callee function (e.g., abs) and compute a function summary.
- For each path in a function, the summary encodes path-condition of each path and the value returned on the path.
- When symbolically executing paths in caller function (e.g., sumAbs) reuse the summary of the callee instead of symbolically executing paths in callee repeatedly.

Compositional Symbolic Execution (contd.)

```
int abs(int x){
  if(x >= 0)
     return x;
  else
    return –x;
}

int sumAbs(int[] a){
   int sum = 0;
   for(int i = 0; i < 50; i++)
      sum += abs(a[i]);
   if(sum == 13)
      error();
   return sum;
}
```

2 paths to symbolically execute

summary of abs function:

forall x. (x ≥ 0 Λ abs(x) = x) V
(x < 0 Λ abs(x) = -x)

No. of paths that lead to error() without descending into abs function = 1

path-condition of path leading to error

abs(a[0]) + abs(a[1]) + …+ abs(a[49]) = 13
Λ forall x. (x ≥ 0 Λ abs(x) = x) V
(x < 0 Λ abs(x) = -x)

Implementation of Symbolic Execution

- Transformation approach
  - transform the program to another program that operates on symbolic values such that execution of the transformed program is equivalent to symbolic execution of the original program
  - difficult to implement, portable solution, suitable for Java, .NET
- Instrumentation approach
  - callback hooks are inserted in the program such that symbolic execution is done in background during normal execution of program
  - easy to implement for C
- Customized runtime approach
  - Customize the runtime (e.g., JVM) to support symbolic execution
  - Applicable to Java, .NET, difficult to implement, flexible, not portable

Implementation of Symbolic Execution for Java (contd.)

```
void foo(int x, int y){
  if(x > y){
    x = x + y;
    y = x – y;
    x = x – y;
    if(x > y)
      assert false;
  }
}
```

original program

```
void foo(Expression x, Expression y){
  if(_GT(x, y)){
    x = _ADD(x, y);
    y = _SUB(x, y);
    x = _SUB(x, y);
    if(_GT(x,y))
      assert false;
  }
}
```

transformed program

```
class Expression{
  int concreteValue;
  Operator op;
  Expression leftOp;
  Expression rightOp;
  …
}
```

Applications of Symbolic Execution

- Test-input generation
- Bug finding
- Program verification
- Determining functional equivalence
- Worst case execution time estimation for real-time software

Limitations of Symbolic Execution

- Limited by the power of constraint solver
  - cannot handle non-linear and very complex constraints
- Does not scale when number of infeasible paths are large. (subject of ongoing research in this area)
- Source code, or equivalent (e.g., Java class files) is required for precise symbolic execution

Path Explosion Problem

```
public void main(string s){
    bool a = contains(s, "Hello");
    bool b = contains(s, "World");
    bool c = contains(s, " at ");
    bool d = contains(s, “GeorgiaTech");
    if (a && b && c && d)
        throw new Exception("found it");
}
```

```
static bool contains(string s, string t){
   if (s == null || t == null)  return false;
   for (int i = 0; i < s.Length-t.Length+1; i++)
      if (containsAt(s, i, t))  return true;
   return false;
}

static bool containsAt(string s, int i, string t){
   for (int j = 0; j < t.Length; j++)
      if (t[j] != s[i + j]) return false;
   return true;
}
```

Complex problem for string-length of 30 characters:

16^{30} possible inputs

383 million execution paths

Summary

- Symbolic execution is a technique for checking feasibility of program paths.
- Feasibility of path is determined by computing path-condition of the path and checking its satisfiability.
- Useful for test-input generation, bug finding, program verification, etc.