WHITE-BOX TESTING -TEST GENERATION

CS3213 FSE

Prof. Abhik Roychoudhury

National University of Singapore

WHAT WE DID EARLIER

- System Requirements: Use-cases, Scenarios, Sequence Diagrams
- System structure: Class diagrams
- Discussion on semantics
- System behavior: State diagrams
- Discussion of the thinking behind your course project
- Static analysis and vulnerability detection: Secure SE
- Software Debugging
- White-box Testing: estimation of a given test-suite

- Today
  - **White-box Testing: automatically generating a test-suite**

Programming Languages — B. Wegbreit, Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

This paper describes the symbolic execution of programs. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. The difficult, yet interesting issues arise during the symbolic execution of conditional branch type statements. A particular system called EFFIGY which provides symbolic execution for program testing and debugging is also described. It interpretively executes programs written in a simple PL/I style programming language. It includes many standard debugging features, the ability to manage and to prove things about symbolic expressions, a simple program testing manager, and a program verifier. A brief discussion of the relationship between symbolic execution and program proving is also included.

Key Words and Phrases: symbolic execution, program testing, program debugging, program proving, program verification, symbolic interpretation
CR Categories: 4.13, 5.21, 5.24

[illegible] one of the [illegible] requirements for applying computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements even when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a program is expected to handle is presented to the program. If the program is judged to produce correct results for the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

Recent work on proving the correctness of programs by formal analysis [5] shows great promise and appears to be the ultimate technique for producing reliable programs. However, the practical accomplishments in this area fall short of a tool for routine use. Fundamental problems in reducing the theory to practice are not likely to be solved in the immediate future.

Program testing and program proving can be considered as extreme alternatives. While testing, a programmer can be assured that sample test runs work correctly by carefully checking the results. The correct execution for inputs not in the sample is still in doubt. Alternatively, in program proving the programmer formally proves that the program meets its specification for all executions without being required to execute the program at all. To do this he gives a precise specification of the correct program behavior and then follows a formal proof procedure to show that the program and the specification are consistent. The confidence in this method hinges on the care and accuracy employed in both the creation of the specification and in the construction of the proof steps, as well as on the attention to machine-dependent issues such as overflow, rounding etc.

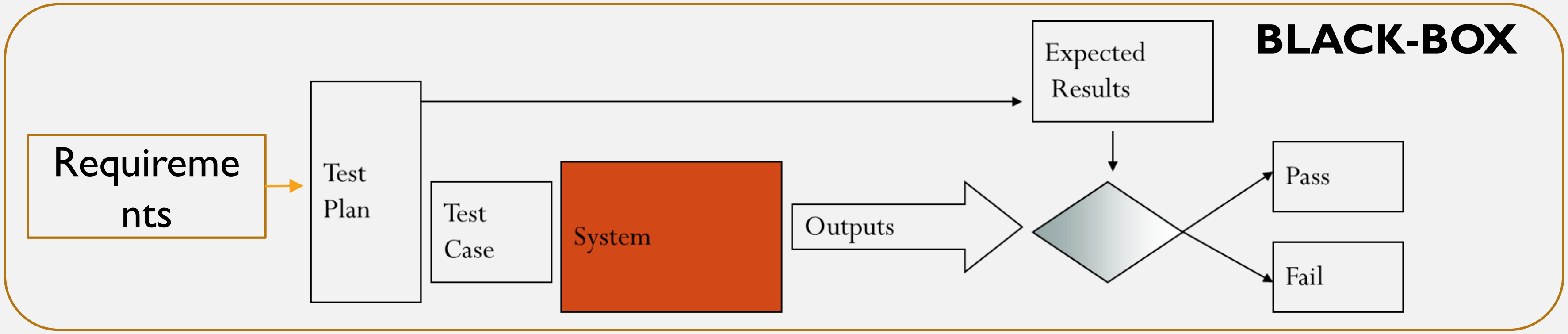
This paper describes a practical approach between these two extremes. From one simple view, it is an enhanced testing technique. Instead of executing a program on a set of sample inputs, a program is "symbolically" executed for a set of *classes* of inputs. That is, each symbolic execution result may be equivalent to a large number of normal test cases. These results can be checked

"Program testing and program proving can be considered as extreme alternatives. ….

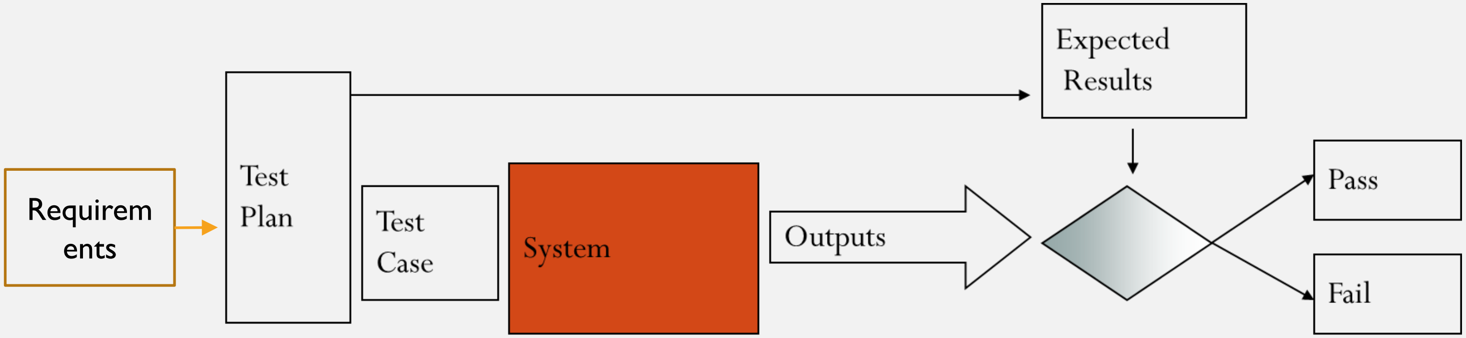
This paper describes a practical approach between these two extremes …

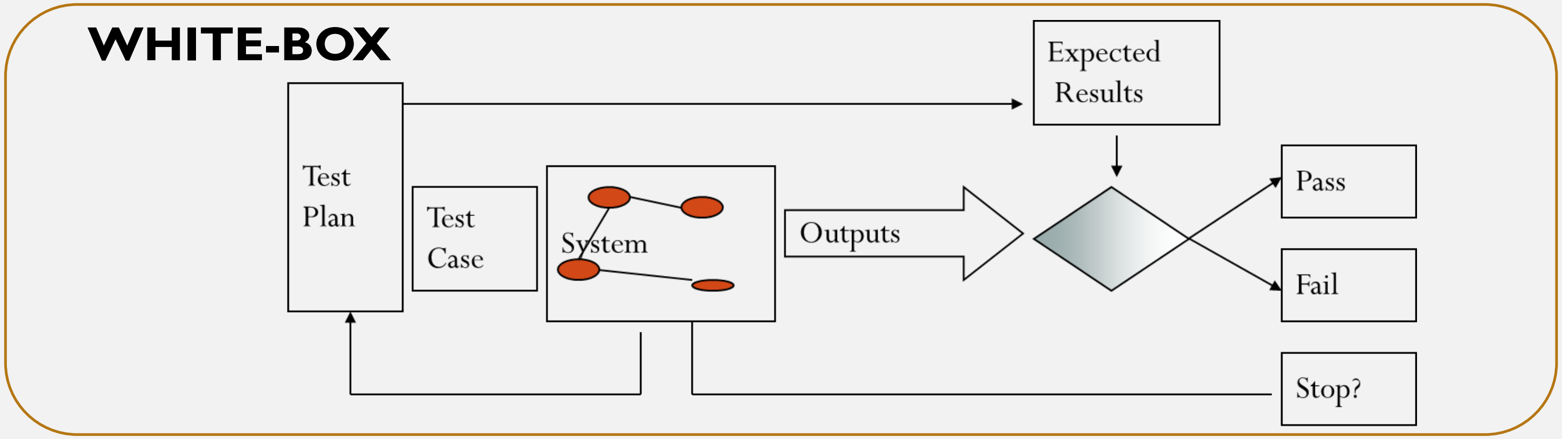
Each symbolic execution result may be equivalent to a large number of normal tests"

TESTING


BLACK-BOX
Requirements
Test Plan
Test Case
System
Outputs
Expected Results
Pass
Fail

TESTING


Requirem
ents
Test
Plan
Test
Case
System
Outputs
Expected
Results
Pass
Fail


WHITE-BOX
Test
Plan
Test
Case
System
Outputs
Expected
Results
Pass
Fail
Stop?

BLURRING THE LINES: SYMBOLIC EXEC.


Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

```
SEARCH( A, L, U, X, found, j){

int j, found = 0;
while (L  <= U  && found == 0){
            j = (L+U)/2;
            if (X == A[j]){  found  = 1;}
            else if (X < A[j]){ U = j -1; }
            else{ L = j +1; }
}
if (found == 0){ j =  L - 1;}

}
```

SEARCH(A, 1, 5, X, found, j)

X == A[3]	found == 1	j == 3
X == A[1] && X < A[3]	found == 1	j == 1
X < A[1] && X <A[3]	found == 0	j == 0
X = A[2] && X > A[1] && X <A[3]	found == 1	j == 2

….

Testing ?
Comprehension??
Verification ???

Programming Languages

B. Wegbreit Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

BLURRING THE LINES: SYMBOLIC EXEC

```
SEARCH( A, L, U, X, found, j){

int j, found = 0;
while (L  <= U  && found == 0){
          j = (L+U)/2;
          if (X == A[j]){  found  = 1;}
          else if (X < A[j]){ U = j -1; }
          else{ L = j +1; }
}
if (found == 0){ j =  L - 1;}

}
```

SEARCH(A, 1, 5, 20, found, j)

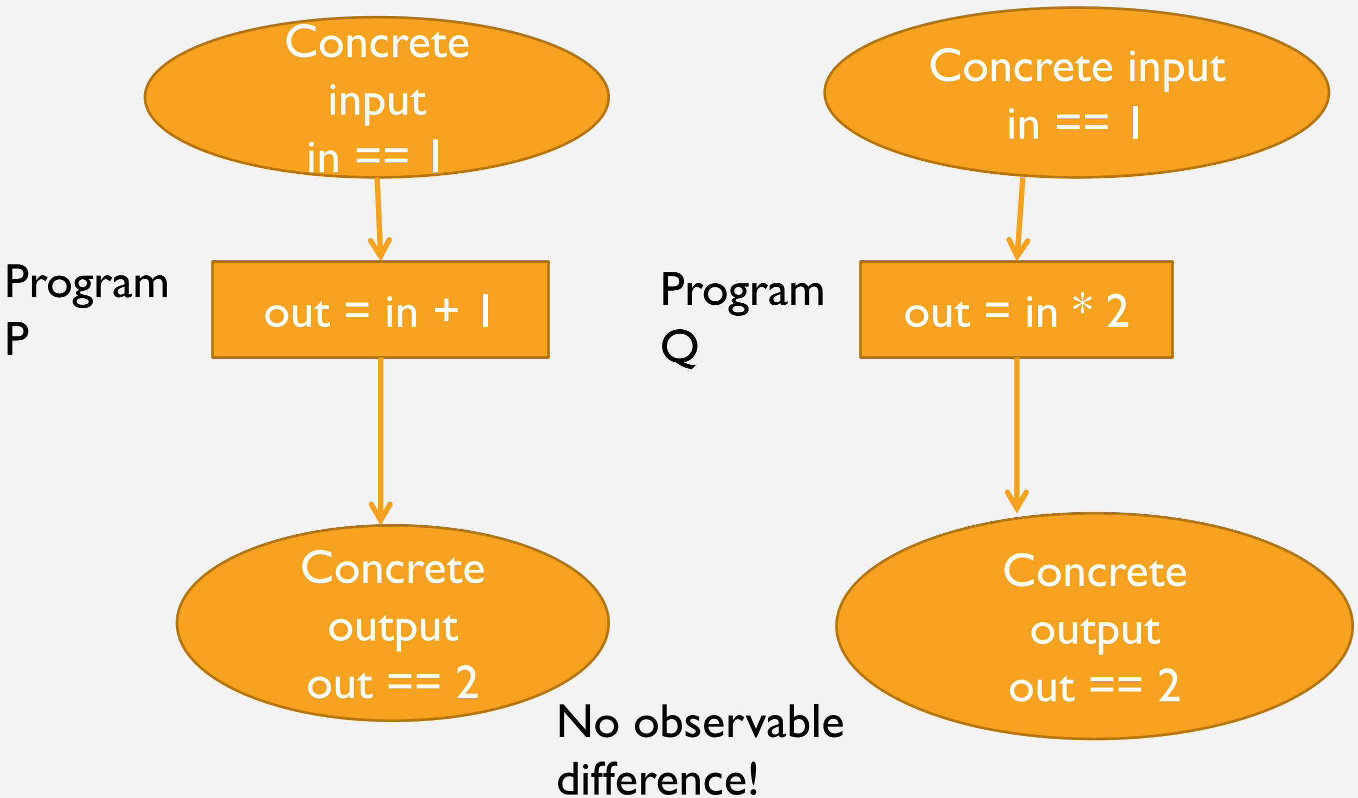
SEARCH(A, 1, 5, X, found, j)

SEARCH(A, N, N+4, X, found, j)

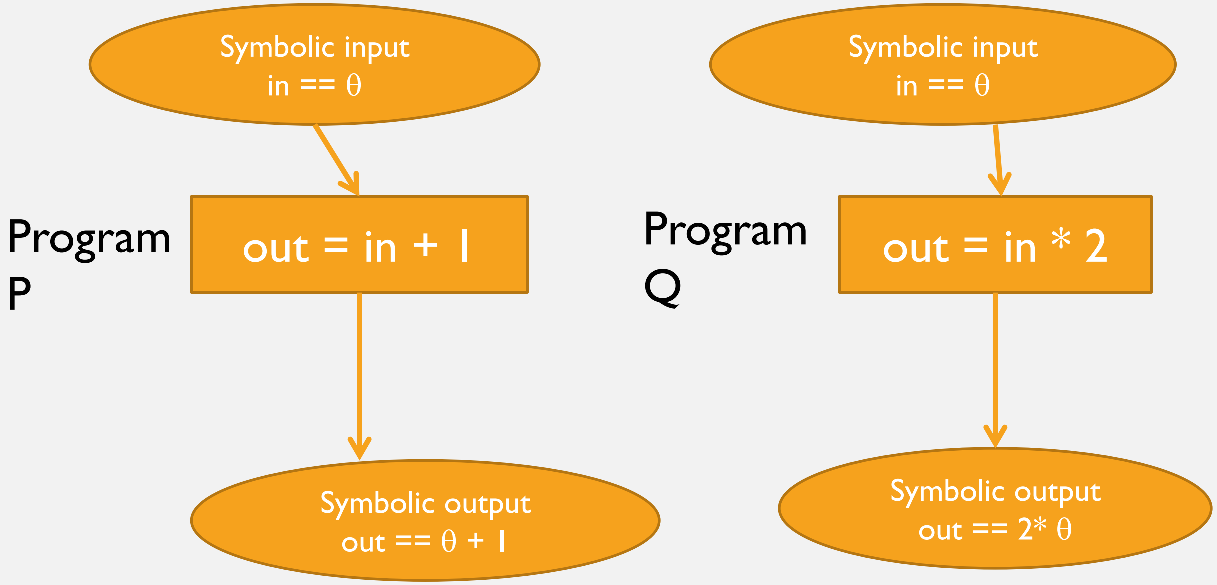
SEARCH(A, 1, M, X, found, j)

Testing ?
Comprehension??
Verification ???

CONCRETE EXECUTION



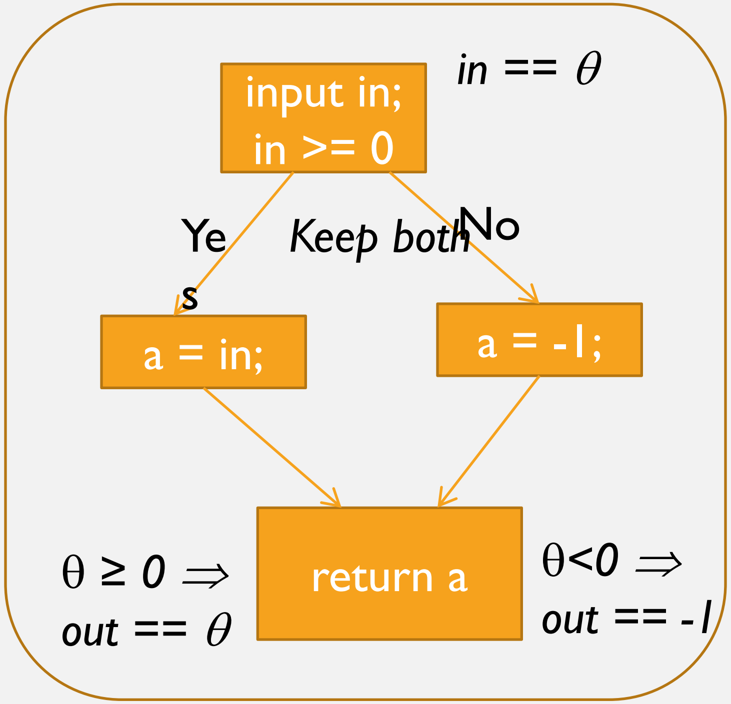
EXECUTION WITH SYMBOLIC INPUTS



To expose difference, try to find θ such that $\theta + 1 \neq 2 * \theta$

PATH EXPLORATION BASED SYMBOLIC EXECUTION

```
input in;

if  (in >= 0)
     a = in;
else
     a = -1;

return a;
```



ON-THE-FLY PATH EXPLORATION

Instead of analyzing the whole program, shift from one program path to another.

in == 0

in == 5

```
input in;
z = 0; x = 0;
if  (in > 0){
    z = in *2;
    x = in +2;
    x = x + 2;
}
else  …
if ( z  > x){
   return error;
}
```

√

X

Sample exploration:

Continue the search for failing inputs. Try those which do not go through the "same" path.

How to perform symbolic execution along a single path?

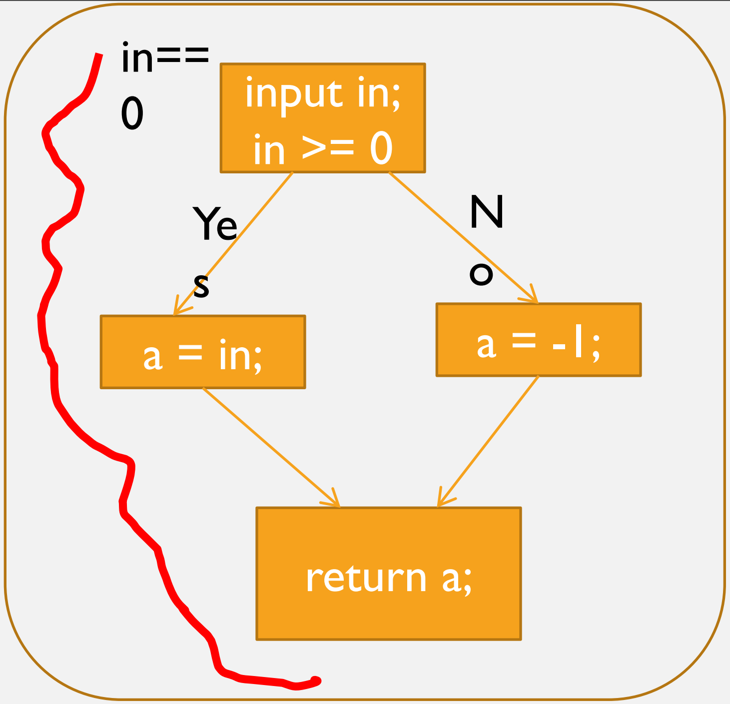
EXPLORING ONE PATH

Useful to find:

"the set of all inputs which trace a given path"

Path condition

in ≥ 0



PATH CONDITION COMPUTATION

in == 5

```
1 input in;
2 z = 0; x = 0;
3 if  (in > 0){
4     z = in *2;
5   x = in +2;
6   x = x + 2;
7  }
8 else  …
9 if ( z  > x){
    return error;
}
```

Line#	Assignment store	Path condition
1	{}	*true*
2	{(z,0),(x,0)}	*true*
3	{(z,0),(x,0)}	*in > 0*
4	{(z,2*in), (x,0)}	*in > 0*
5	{(z,2*in), (x,in+2)}	*in > 0*
6	{(z,2*in), (x, in+4)}	*in > 0*
7	{(z, 2*in), (x, in+4)}	*in > 0*
9	{(z, 2*in), (x, in+4)}	$in>0 \wedge (2*in > in + 4)$

DIRECTED TESTING

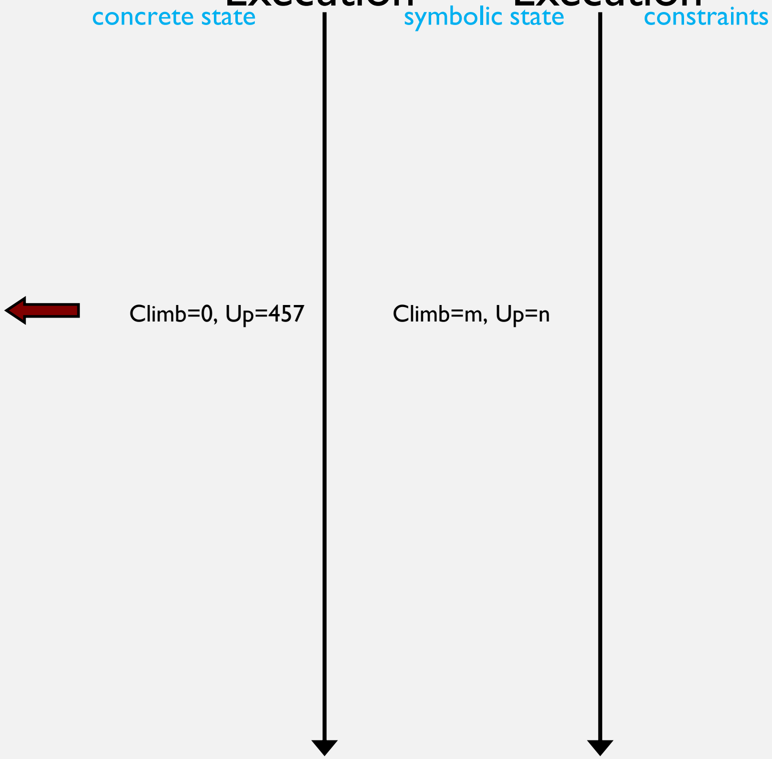
- Start with a random input I.
- Execute program P with I
  - Suppose I executes path p in program P.
  - While executing p, collect a symbolic formula f which captures the set of all inputs which execute path p in program P.
  - **f is the path condition of path p traced by input i.**
- Minimally change f, to produce a formula f1
  - Solve f1 to get a new input I1 which executes a path p1 different from path p.

```
main(){
        int t1 = randomInt();
        int t2 = randomInt();
        test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
      sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
       upward = 1;
    } else {upward = 0;}
    if (upward < 0){
           abort;
     } else return upward;
}
```

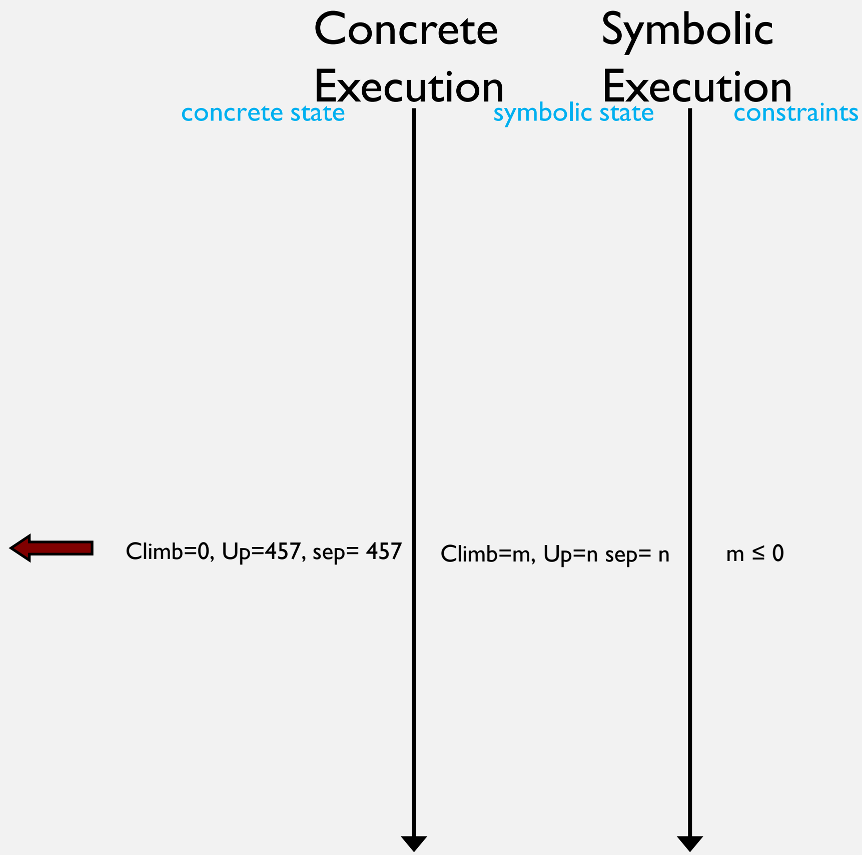
Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
t1=0, t2=457		t1=m, t2=n	

Concrete Execution   Symbolic Execution

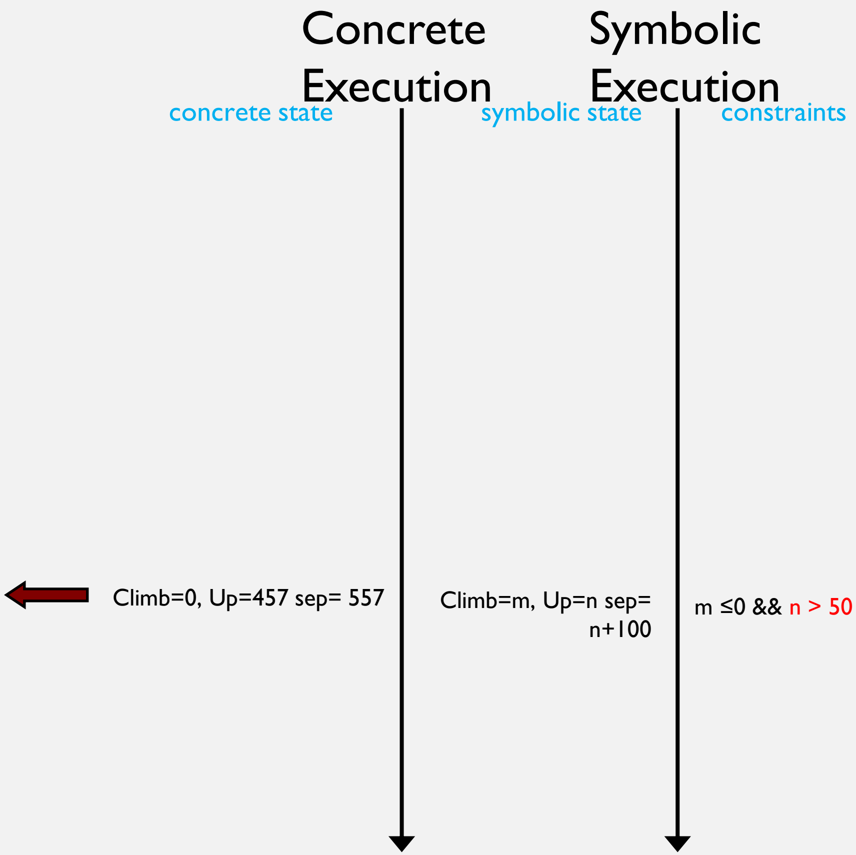
```
main(){
        int t1 = randomInt();
        int t2 = randomInt();
        test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
      sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
       upward = 1;
    } else {upward = 0;}
    if (upward < 0){
           abort;
     } else return upward;
}
```



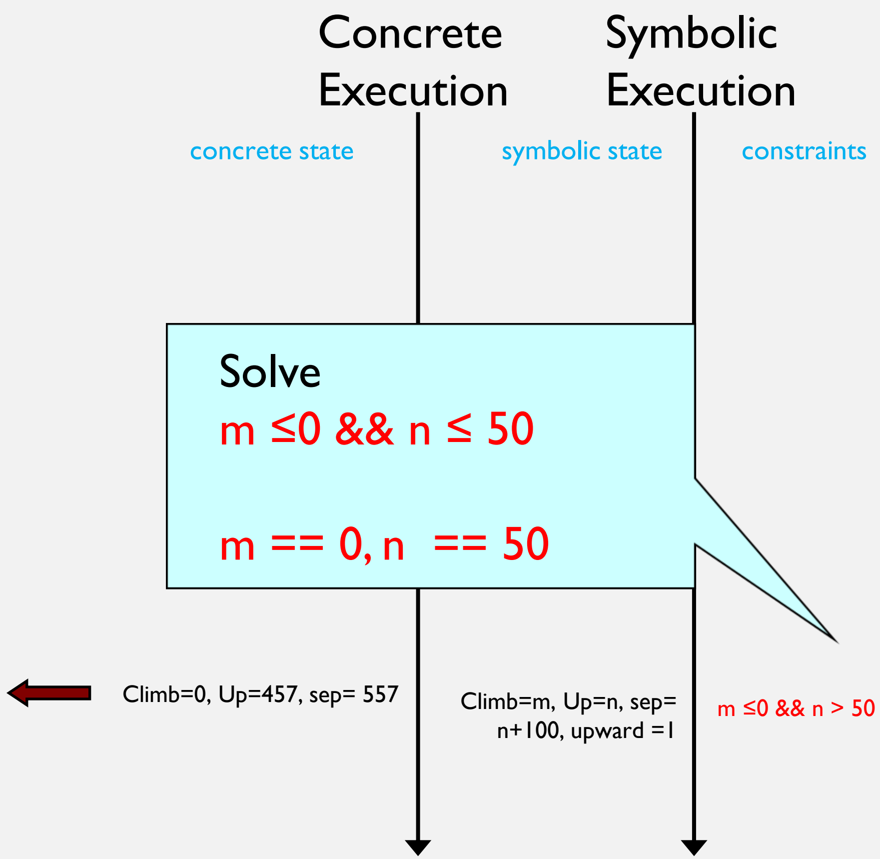
```
main(){
        int t1 = randomInt();
        int t2 = randomInt();
        test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
      sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
       upward = 1;
    } else {upward = 0;}
    if (upward < 0){
          abort;
     } else return upward;
}
```



```
main(){
        int t1 = randomInt();
        int t2 = randomInt();
        test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb){
      sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
       upward = 1;
    } else {upward = 0;}
    if (upward < 0){
           abort;
     } else return upward;
}
```



```
main(){
        int t1 = randomInt();
        int t2 = randomInt();
        test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb){
      sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
       upward = 1;
    } else {upward = 0;}
    if (upward < 0){
          abort;
     } else return upward;
}
```



Ack: Koushik Sen (Berkeley)

Concrete Execution

Symbolic Execution

concrete state

symbolic state

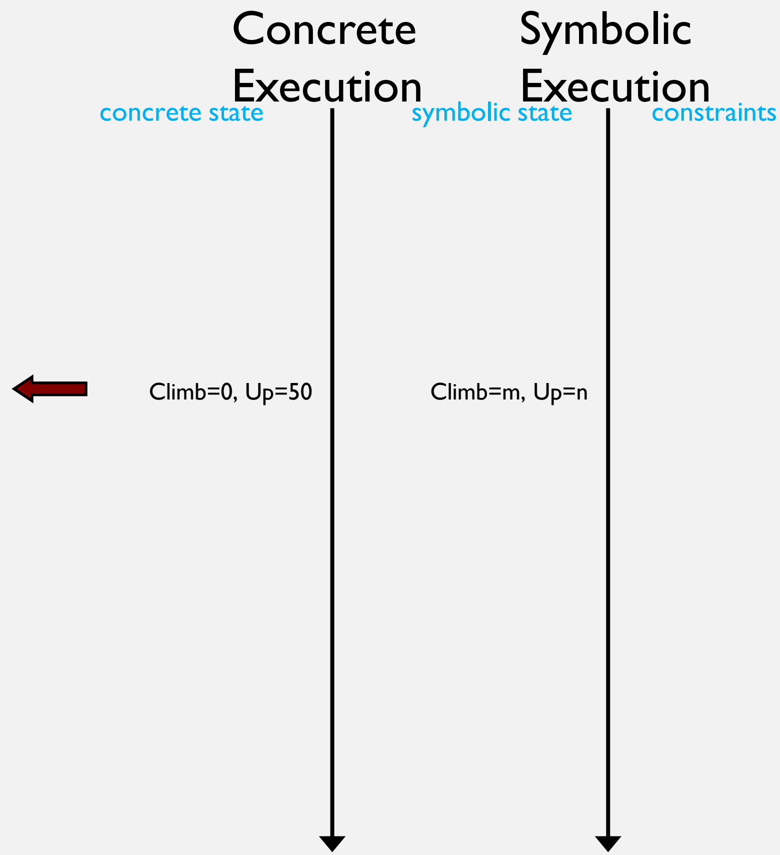
constraints

```
main(){
        int t1 = randomInt();
        int t2 = randomInt();
        test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
      sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
       upward = 1;
    } else {upward = 0;}
    if (upward < 0){
           abort;
     } else return upward;
}
```

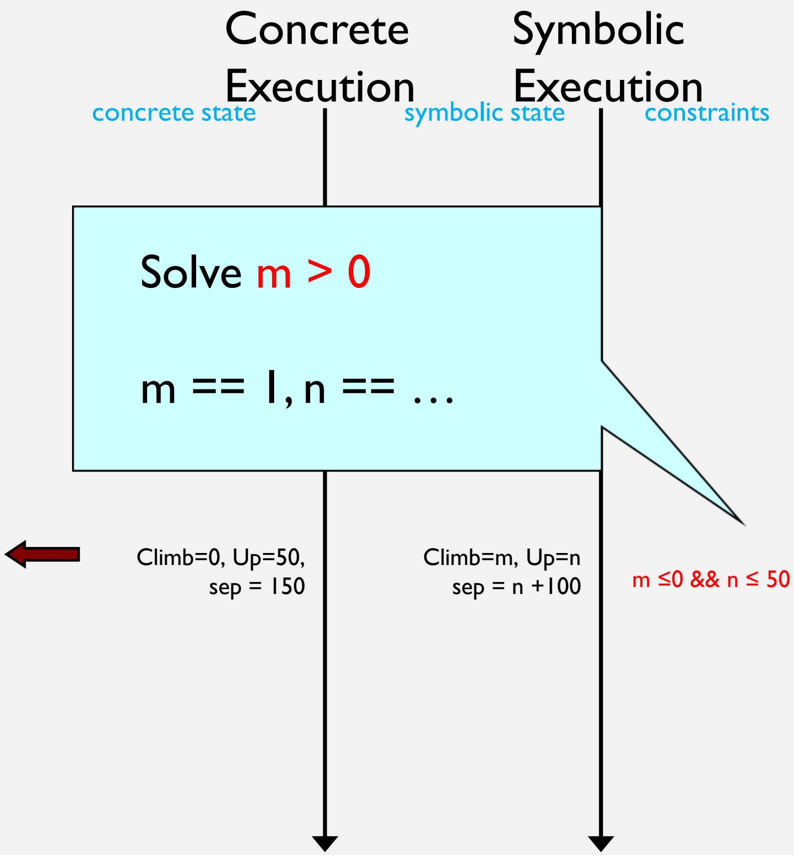
t1=0, t2=50

t1=m, t2=n

```
main(){
        int t1 = randomInt();
        int t2 = randomInt();
        test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
      sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
       upward = 1;
    } else {upward = 0;}
    if (upward < 0){
           abort;
     } else return upward;
}
```

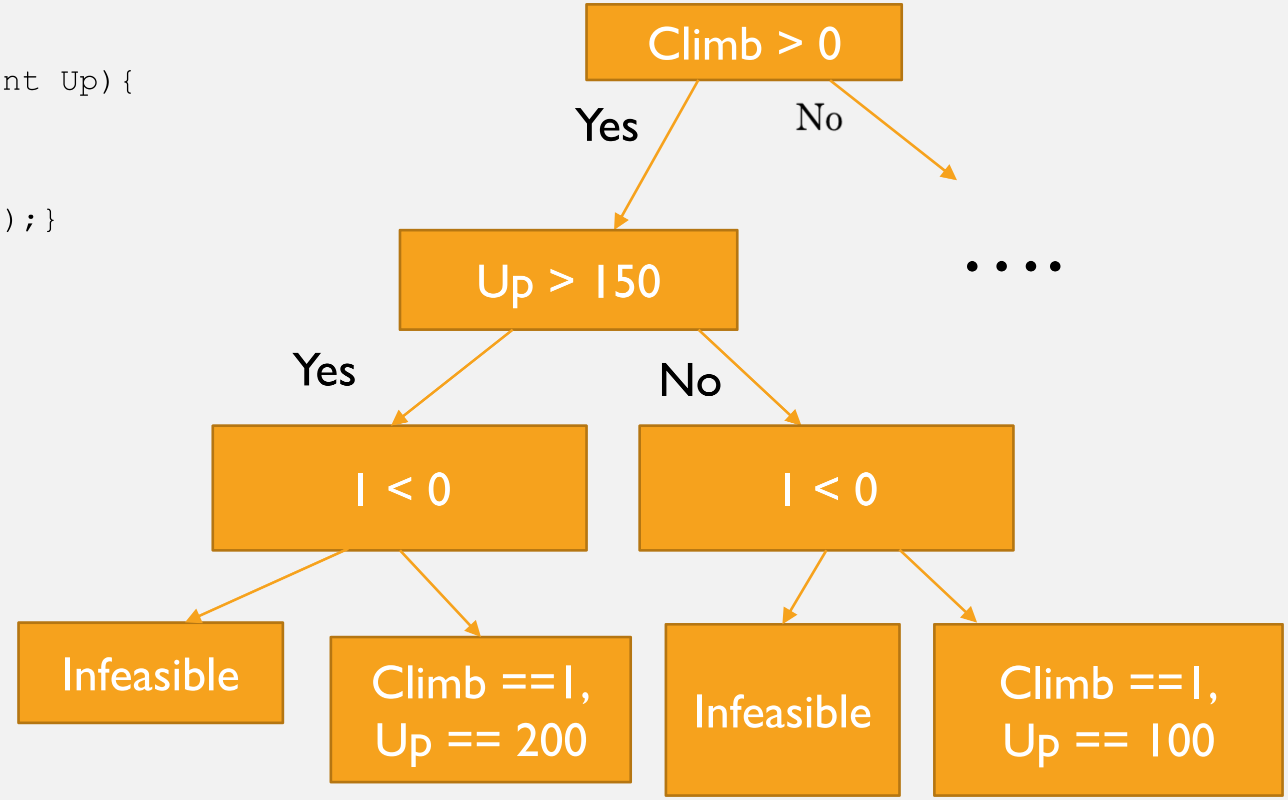


```
main(){
        int t1 = randomInt();
        int t2 = randomInt();
        test_me(t1,t2);
}
int add100(int x){ return x + 100;}

int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
      sep = Up;}
  else  {sep = add100(Up);}
   if (sep > 150){
       upward = 1;
    } else {upward = 0;}
    if (upward < 0){
            abort;
     } else return upward;
}
```

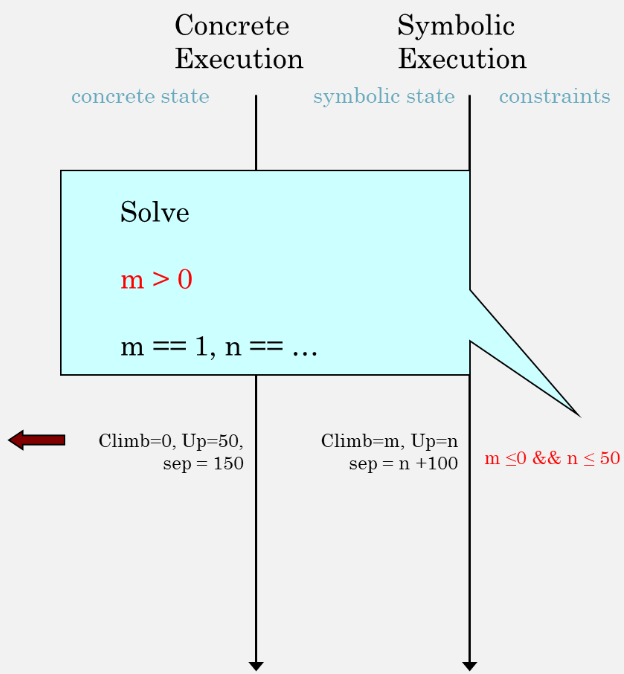


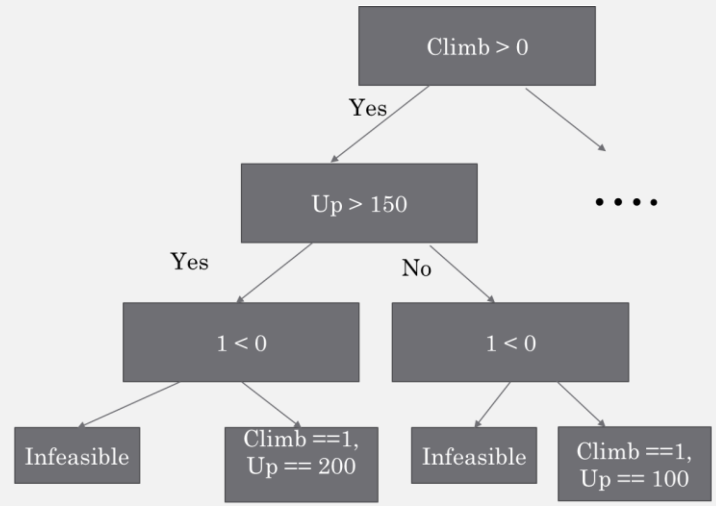
SYMBOLIC EXECUTION TREE

```
int test_me(int Climb, int Up){
  int sep, upward;
  if (Climb > 0){
      sep = Up;}
  else  {sep = add100(Up);}
  if (sep > 150){
       upward = 1;
    } else {upward = 0;}
  if (upward < 0){
           abort;
  } else return upward;
}
```



CONCOLIC AND SYMBOLIC





One path at a time, simplify constraints! Strategies!!

Entire execution tree, Search

SYMBOLIC AND CONCOLIC

- Symbolic
  - Execute IF(r)/then/else :fork [provided r is unresolved]
    - Then: PC := PC $\wedge$ r AND
    - Else: PC := PC $\wedge$ $\neg$r

- Concolic:
  - Execute IF(r)
    - Resolved branch condition r using concrete values
    - Suppose true, PC := PC $\wedge$ r , OR
    - Suppose false, PC := PC $\wedge$ $\neg$r

DART IN A NUTSHELL

- Dynamically observe random execution and generate new test inputs to drive the next execution along an alternative path
  - do dynamic analysis on a random execution
  - collect symbolic constraints at branch points
  - negate one constraint at a branch point (say b)
  - call constraint solver to generate new test inputs
  - use the new test inputs for next execution to take alternative path at branch b
  - (Check that branch b is indeed taken next)

ADVANTAGE OF DYNAMIC ANALYSIS OVER STATIC ANALYSIS

```
struct foo { int i; char c; }

bar (struct foo *a) {
    if (a->c == 0) {
        *((char *)a + sizeof(int)) = 1;
        if (a->c != 0) {
            abort();
        }
    }
}
```

- Reasoning about dynamic data is easy
- Due to limitation of alias analysis "static analyzers" cannot determine that "a->c" has been rewritten
- DART finds the error
  - sound

DISCUSSION

- In comparison to existing testing tools, DART is
  - light-weight
  - dynamic analysis (compare with static analysis)
    - ensures no false alarms
  - concrete execution and symbolic execution run simultaneously
    - symbolic execution consults concrete execution whenever dynamic analysis becomes intractable
  - real tool that works on real C programs
    - completely automatic

JUST CHECKING

- .. *Whether we are all awake (a bit late in the day !)*

- Consider two programs P1, P2 both of which take integer inputs x, y and produce integer output z.

- P1: if (x > y){ z = x + y; if (z > x){ z = z+1;}} else{z = x – y;}
- P2: if (x < y){z = x – y;} else{ z = x + y;}

- Construct a logical formula which captures all test inputs which generate different outputs in P1 and P2.

Answer: The path summaries in P1 are

$x \leq y \Rightarrow z == x - y$
$x > y \wedge y > 0 \Rightarrow z == x + y + 1$
$x > y \wedge y \leq 0 \Rightarrow z == x + y$

The path summaries in P2 are
$x < y \Rightarrow z == x - y$
$x \geq y \Rightarrow z == x + y$

By comparing the two path summaries we see that the output expressions are different when $x == y$ and when $x > y > 0$

Scenario 1: when $x == y$, P1 returns $x - y$ and P2 returns $x + y$ These two expressions are unequal when y != 0. So, this is captured by the constraint

$y \neq 0 \wedge x == y$

Scenario 2: when $x > y > 0$, P1 returns $x + y + 1$ and P2 returns $x + y$ These two expressions are never equal. So, we get the constraint

$x > y > 0$

Overall, the set of test inputs producing different outputs in the two programs are captured by the formula

$(x > y > 0) \vee (y \neq 0 \wedge x == y)$