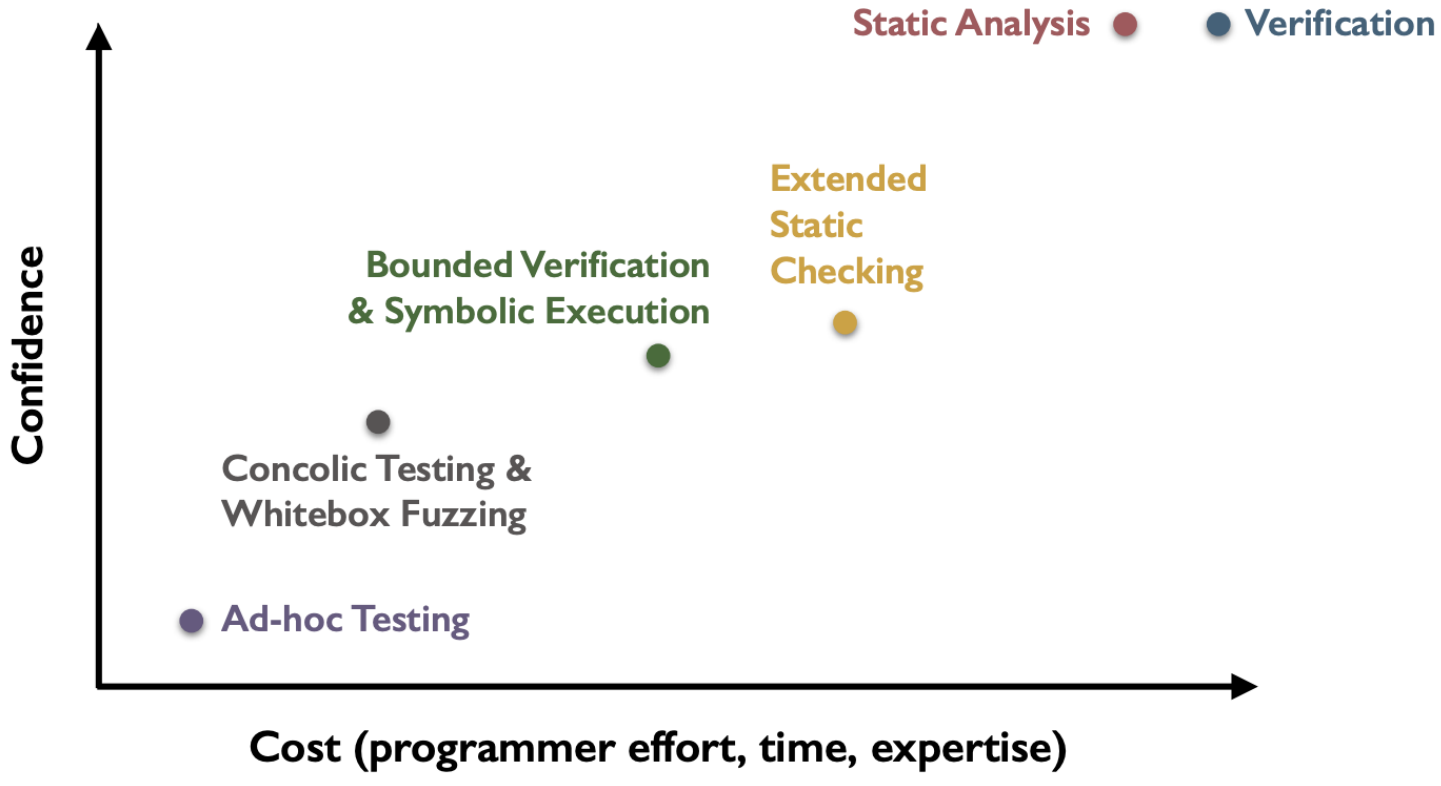
CMSC 433
Programming Language Technologies and Paradigms

Concolic Execution

Spectrum of program validation methods



Path explosion

How to test the following program effectively?

```
foo(a,b,c){
  x = 0;
  y = 0;
  if(a > 1){x = 1;}
  else{x = 10;}

  if(b > 10){y = 2;}
  else{y = 20;}

  if(c > 20){z = 3;}
  else{z = 30;}
  return (x + y) + z;
}
```

With nested 'if' statements of depth N, the number of possible paths are $O(2^N)$.

1:[a > 1, b <= 10, c <= 20]
2:[a > 1, b > 10, c > 20]
3:[a <= 1, b > 10, c > 20]
4:[a <= 1, b <= 10, c <= 20]
5:[a <= 1, b > 10, c <= 20]
6:[a > 1, b > 10, c <= 20]
7:[a <= 1, b <= 10, c > 20]
8:[a > 1, b <= 10, c > 20]

Environment modeling

```
foo(n){
  if(e){
    x = lib(n);  // lib() is a library function:
  }else{
    x = 2*n-4;
  }
  return 1/x;
}
```

- If we don't establish a symbolic model for the "lib()" function, we cannot check the true branch symbolically.
- But we can still executes that branch concretely!
- So the symbolic model for the library function is just to improve the precision!

Solver limitation

```
foo(x, y){
  int m = x*x*x;
  if(m == y){
    assert(…);
  }else{
    …;
  }
  return 1;
}
```

Path condition x*x*x == y is generally beyond the capability of many modern solvers.

Concolic execution

- Concolic = Concrete + symbolic

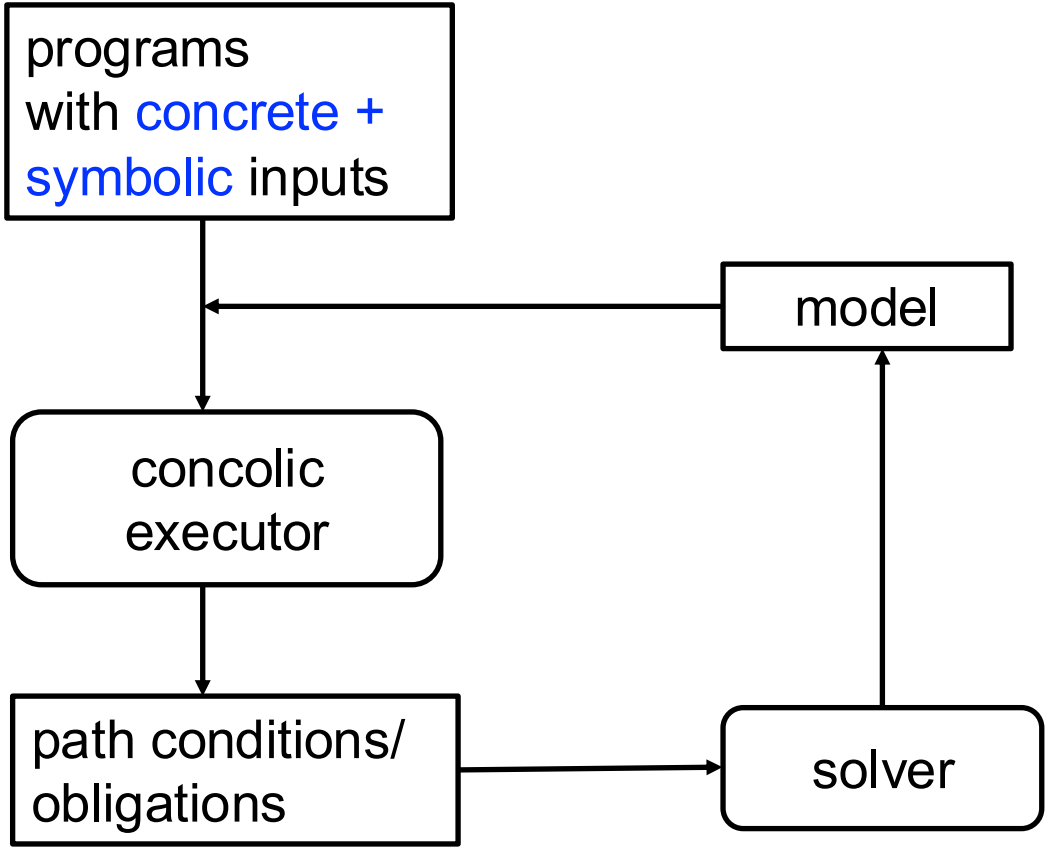
- Initially developed around 2005:
  - *DART: Directed Automated Random Testing*, by Patrice Godefroid; Nils Klarlund; Koushik Sen, 2005

  - *CUTE: a concolic unit testing engine for C*, by Koushik Sen; Darko Marinov; Gul Agha, 2005

Concolic Execution Steps

- 1: generate random concrete input and symbolic input
- 2: run program with the *two* inputs
  - maintain the concrete+symbolic states
- 3: For branching, generate path conditions
  - just like in the symbolic execution
  - but don’t fork(), only go to feasible path
- 4: After one run, negate the result Path Conditions, send them to solver, to get other concreate input
- Go to 2, re-run the program

Architecture



Concolic execution

Step 1: construct random values as initial input:

```
foo(x, y){
  z = x+y;
  if(x == 32467289)
    return x/y;
  else
    return 0;
}
```

Variable	value	sym.value
x	1	x
y	1	y

Concolic execution

Step 2: execute the program with both the concrete and symbolic values, maintaining the program state:

```
foo(x, y){
  z = x+y;
  if(x == 32467289)
    return x/y;
  else
    return 0;
}
```

Variable	value	sym.value
x	1	x
y	1	y
z	2	x+y

Concolic execution

For branch, the executor branches according to the concreate values, and we add a “path condition” to the target branch:

```
foo(x, y){
  z = x+y;
  if(x == 32467289)
    return x/y;
  else
    return 0;
}
```

Variable	value	sym.value
x	1	x
y	1	y
z	2	x+y

```
x != 32467289
```

Concolic execution

When finish and get the path conditions, we negate the path conditions and send it to solver, to get new inputs:

```
foo(x, y){
  z = x+y;
  if(x == 32467289)
    return x/y;
  else
    return 0;
}
```

Variable	value	sym.value
x	32467289	x
y	1	y
z	32467290	x+y

```
x == 32467289
```

Concolic execution

With the new path condition and obligations, we can generate final constraints:

```
foo(x, y){
  z = x+y;
  if(x == 32467289)
    return x/y;
  else
    return 0;
}
```

Variable	value	sym.value
x	32467289	x
y	1	y
z	32467290	x+y

```
x == 32467289
```

The general form

Conceptually, we negate one of PCs one time:

```
int foo(int x, int y){
  if(b1){  b1
    if(b2){   b1 ∧b2
              return x/y;
    }else …;{}
  }else{
    return 0;
  }
}
```

Practical Issues of Symbolic Execution

- Path explosion
- Loops and recursions
- Environment modeling

Path explosion

- In concolic execution, the number of paths explored can be controlled, according to metrics such as:
  - the total number
  - the coverage
  - the timeout
  - etc.

Loops and recursions

Loops introduce non-termination:

```
sum(int n){
  s = 0, i = 0;
  while(i<=n){
    s = s+i;
    i = i+1;
  }
  return s;
}
```

Loops and recursions

No need to finitize the loops.

```
sum(n){
  s = 0, i = 0;
  while(i<=n){
    s = s+i;
    i = i+1;
  }
  return s;
}
```

Environment modeling

```
foo(n){
  if(e){
    x = lib(n); //lib() is a library function:
  }else{
    x = 3*n;
  }
  return 1/x;
}
```

- If we don’t establish a symbolic model for the “lib()” function, we cannot symbolic check the true branch.
- But we can still concolic executes that branch!

Summary

- Concolic execution is a more practical (and flexible) infrastructure for program testing
  - sacrifice completeness for usability
- Many practical tools with many applications
  - with many improvements