Symbolic Execution

Subhajit Roy

Indian Institute of Technology Kanpur

Outline

Correctness of Software

- Program Verification (sound, incomplete, infinite inputs)

[1] The testing community often uses different definitions of soundness and completeness

Correctness of Software

- Program Verification (sound, incomplete, infinite inputs)
- Program Testing[1] (unsound, complete, finite inputs)

[1] The testing community often uses different definitions of soundness and completeness

Correctness of Software

- Program Verification (sound, incomplete, infinite inputs)
- Program Testing[1] (unsound, complete, finite inputs)
- Symbolic Execution (sound, complete, infinite inputs)

[1] The testing community often uses different definitions of soundness and completeness

Symbolic Execution

Analyse this

What inputs cause this program to violate the assertion?

```
int main(){

 input(a,b,c,d);
 if ( a <= b){
    c++;
 }
 else {
    d++;
    if ( c == 2*d)
         assert(a > d)
 }
}
```

Symbolic Execution

Analyze this

OK, let's answer this!

Symbolic Execution

Analyze this

OK, let's answer this!

A customer buys 5 hot dogs and 5 bags of potato chips for \$12.50. Another customer buys 3 hot dogs and 4 bags of potato chips for \$8.25. Find the cost of each item.[a]

Symbolic Execution

Analyze this

OK, let's answer this!

A customer buys 5 hot dogs and 5 bags of potato chips for \$12.50. Another customer buys 3 hot dogs and 4 bags of potato chips for \$8.25. Find the cost of each item.[a]

== Use symbols to represent unknowns! ==

[a] https://www.wyzant.com/resources/answers/107505/fond_the_cost_of_each_item

Symbolic Execution

Simple idea

Execute a program with symbolic inputs!

Symbolic Execution

Simple idea

Execute a program with symbolic inputs!

```
int main(){

 input(a,b,c,d);
 if ( a <= b){
    c++;
 }
 else {
    d++;
    if ( c == 2*d)
        assert(a > d)
 }
}
```

Symbolic Execution

Simple idea

Execute a program with symbolic inputs!

```
int main(){

 input(a,b,c,d);
 if ( a <= b){
    c++;
 }
 else {
    d++;
    if ( c == 2*d)
        assert(a > d)
 }
}
```

VariableName	SymbolicName
a	$\alpha 0$
b	$\alpha 1$
c	$\alpha 2$
d	$\alpha 3$

Symbolic Execution

Simple idea

Execute a program with symbolic inputs!

```
int main(){

 input(a,b,c,d);
 if ( a <= b){
    c++;
 }
 else {
    d++;
    if ( c == 2*d)
        assert(a > d)
 }
}
```

Analyze the **Path Condition**

VariableName	SymbolicName
a	$\alpha 0$
b	$\alpha 1$
c	$\alpha 2$
d	$\alpha 3$

Bounded Model Checking v/s Symbolic Execution

BMC

$$\phi_k = STEP(X_0, X_1) \wedge STEP(X_1, X_2) \cdots \wedge STEP(X_{k-1}, X_k)$$

SE

$$\phi_k = STEP(X_0, X_1) \wedge STEP(X_1, X_2) \cdots \wedge STEP(X_{k-1}, X_k)$$

Bounded Model Checking v/s Symbolic Execution

BMC

$$\phi_k = STEP(X_0, X_1) \wedge STEP(X_1, X_2) \cdots \wedge STEP(X_{k-1}, X_k)$$

SE

$$\phi_k = STEP(X_0, X_1) \wedge STEP(X_1, X_2) \cdots \wedge STEP(X_{k-1}, X_k)$$

... but on a single path!

Bounded Model Checking v/s Symbolic Execution

BMC

$$\phi_k = STEP(X_0, X_1) \wedge STEP(X_1, X_2) \cdots \wedge STEP(X_{k-1}, X_k)$$

SE

$$\phi_k = STEP(X_0, X_1) \wedge STEP(X_1, X_2) \cdots \wedge STEP(X_{k-1}, X_k)$$

... but on a single path!

SE can be seen as performing BMC on each path in isolation.

Gaining Coverage

How to explore multiple (potentially, all) paths:

- **Concolic execution** Concrete execution on random input, collect constraints, edit constraints, solve for new path

Gaining Coverage

How to explore multiple (potentially, all) paths:

- **Concolic execution** Concrete execution on random input, collect constraints, edit constraints, solve for new path
- **EGT (Execution generated testing)** Symbolically execute, fork at branches

EGT

```
int main(){

 input(a,b,c,d);
 symbolic(a,b,c,d);
 if ( a <= b){
    c++;
    if ( c <= d)
      printf("Hi\n");
 }
 else {
    d++;
    if ( c*c == d)
      printf("Bye\n");
 }
}
```

EGT

```
int main(){

 input(a,b,c,d);
 symbolic(a,b,c,d);
 if ( a <= b){
    c++;
    if ( c <= d)
      printf("Hi\n");
 }
 else {
    d++;
    if ( c*c == d)
      printf("Bye\n");
 }
}
```

VariableName	SymbolicName
a	$\alpha 0$
b	$\alpha 1$
c	$\alpha 2$
d	$\alpha 3$

EGT

Covering behaviors

Fork at each branch if both sides are feasible (use an SMT solver)

```
int main(){

 input(a,b,c,d);
 symbolic(a,b,c,d);
 if ( a <= b){
    c++;
    if ( c <= d)
       printf("Hi\n");
 }
 else {
    d++;
    if ( c*c == d)
       printf("Bye\n");
 }
}
```

PC:TRUE
Line:3-5

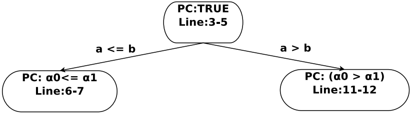
EGT

Covering behaviors

Fork at each branch if both sides are feasible (use an SMT solver)

```
int main(){

 input(a,b,c,d);
 symbolic(a,b,c,d);
 if ( a <= b){
    c++;
     ⟶if ( c <= d)
       printf("Hi\n");
 }
 else {
    d++;
    ⟶if ( c*c == d)
       printf("Bye\n");
 }
}
```



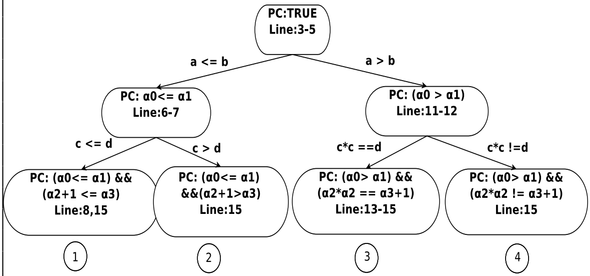
EGT

Generating test

Solve the *path condition* (PC) to synthesize testcases

```
int main ( ) {
int main(){

 input(a,b,c,d);
 symbolic(a,b,c,d);
 if ( a <= b){
    c++;
    if ( c <= d)
      printf("Hi\n");
 }
 else {
    d++;
    if ( c*c == d)
      printf("Bye\n");
 }
⟶}
```



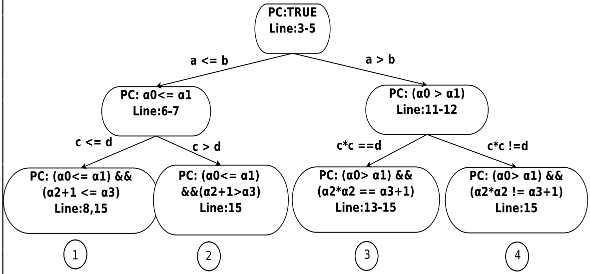
EGT

Generating test

Solve the *path condition* (PC) to synthesize testcases

```
int main ( ) {
int main(){

 input(a,b,c,d);
 symbolic(a,b,c,d);
 if ( a <= b){
    c++;
    if ( c <= d)
       printf("Hi\n");
 }
 else {
    d++;
    if ( c*c == d)
       printf("Bye\n");
 }
⟶}
```



Path	PC	Assignment
1	$(\alpha_0 <= \alpha_1) \wedge (\alpha_2 + 1 <= \alpha_3)$	(0,1,2,3)
2	$(\alpha_0 <= \alpha_1) \wedge (\alpha_2 + 1 > \alpha_3)$	(0,1,4,3)
3	$(\alpha_0 > \alpha_1) \wedge (\alpha_2 * \alpha_2 == \alpha_3 + 1)$	(1,0,2,3)
4	$(\alpha_0 > \alpha_1) \wedge (\alpha_2 * \alpha_2 \neq \alpha_3 + 1)$	(1,0,0,0)

Concolic Testing

1. Run program on a random input
2. Collect the *path condition* as the program executes
3. Modify the path condition when program terminates
4. Solve modified path condition to generate new input for the next run of the program

Handling real-world programs

- external function calls
- vector instructions
- system calls
- floating-point instructions
- non-linear arithmetic ...

Handling real-world programs

- external function calls
- vector instructions
- system calls
- floating-point instructions
- non-linear arithmetic ...

== *Concretization* and *Virtualization* ==

Concretization

```
int main ( ) {
  read ( x ) ;
  if ( x > 0) {
   y = foo ( x );
   if ( y > 150)
     print ("less");
   if ( y > 250)
     assert (0);
   if ( y != a )
     assert (0);
   if ( y < 0)
     assert (0);
   }
}
```

Concretization

```
int main ( ) {
  read ( x ) ;
  if ( x > 0) {
   y = foo ( x );
   if ( y > 150)
     print ("less");
   if ( y > 250)
     assert (0);
   if ( y != a )
     assert (0);
   if ( y < 0)
     assert (0);
   }
}
```

Possible solutions

Concretization

```
int main ( ) {
  read ( x ) ;
  if ( x > 0) {
   y = foo ( x );
   if ( y > 150)
     print ("less");
   if ( y > 250)
     assert (0);
   if ( y != a )
     assert (0);
   if ( y < 0)
     assert (0);
   }
}
```

Possible solutions

- Overapprox

  $y \leftarrow *$

Concretization

```
int main ( ) {
  read ( x ) ;
  if ( x > 0) {
   y = foo ( x );
   if ( y > 150)
     print ("less");
   if ( y > 250)
     assert (0);
   if ( y != a )
     assert (0);
   if ( y < 0)
     assert (0);
   }
}
```

Possible solutions

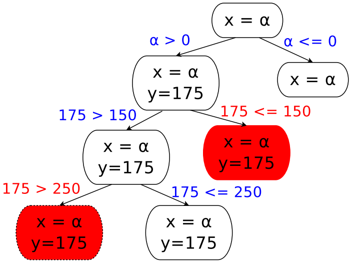
- Overapprox
  $y \leftarrow *$
- Underapprox (*concretization*)
  $y \leftarrow 0$

Problems with concretization

```
int main ( ) {
  read ( x ) ;
  if ( x > 0){
    y = foo(x);
    if ( y > 150)
      print("less");
    if ( y > 250)
      assert (0);
    if ( y != x )
      assert (0);
    if ( y < 0)
      assert (0);
  }
}
```

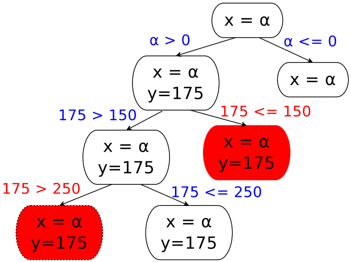
Problems with concretization

```
int main ( ) {
  read ( x ) ;
  if ( x > 0){
    y = foo(x);
    if ( y > 150)
      print("less");
    if ( y > 250)
      assert (0);
    if ( y != x )
      assert (0);
    if ( y < 0)
      assert (0);
   }
}
```



Problems with concretization

```
int main ( ) {
  read ( x ) ;
  if ( x > 0){
    y = foo(x);
    if ( y > 150)
      print("less");
    if ( y > 250)
      assert (0);
    if ( y != x )
      assert (0);
    if ( y < 0)
      assert (0);
  }
}
```



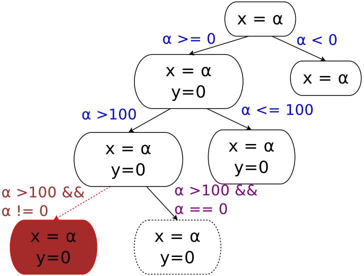
Loss in coverage! (unsound)

Problems with concretization

```
int main ( ) {
  read(x);
  if ( x >= 0) {
→ y = abs(x);
    if (x > 100)
      if ( y != x )
        assert (0);
  }
}
```

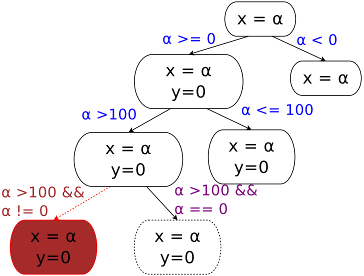
Problems with concretization

```
int main ( ) {
  read(x);
  if ( x >= 0) {
→ y = abs(x);
    if (x > 100)
      if ( y != x )
        assert (0);
  }
}
```



Problems with concretization

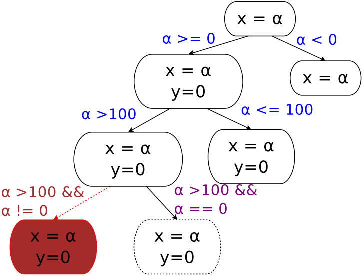
```
int main ( ) {
  read(x);
  if ( x >= 0) {
→ y = abs(x);
    if (x > 100)
      if ( y != x )
        assert (0);
  }
}
```



False positive! (incompleteness—in a testing tool?)

Problems with concretization

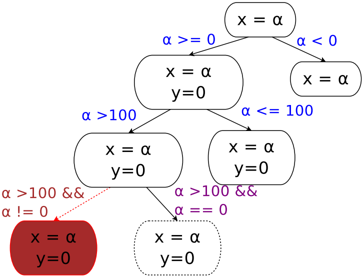
```
int main ( ) {
  read(x);
  if ( x >= 0) {
→ y = abs(x);
    if (x > 100)
      if ( y != x )
        assert (0);
  }
}
```



False positive! (incompleteness—in a testing tool?)
Is it a bug? (no, a conscious design decision)

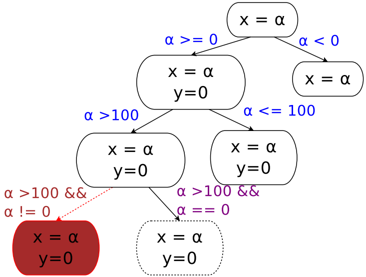
Problems with concretization

```
int main ( ) {
  read ( x ) ;
  if ( x >= 0){
→   y = abs(x);
    if (x > 100)
      if ( y != x )
        assert (0);
    }
}
```



Problems with concretization

```
int main ( ) {
  read ( x ) ;
  if ( x >= 0){
→   y = abs(x);
    if (x > 100)
      if ( y != x )
        assert (0);
    }
}
```



Reproducibility (of tests)?

Question

Can we regain **soundness**, **completeness** and **reproducibility** lost due to concretizations?

Question

Can we regain **soundness**, **completeness** and **reproducibility** lost due to concretizations?

Pandey, Kotcharlakota and Roy. Deferred concretization in symbolic execution via fuzzing. ISSTA 2019.

Modeling heap memory

```
int main ( ) {
  read(x);
  a = malloc(100);
  clear(a);
  a[x] = 5
  assert(a[x] != a[2*x - 2]);
}
```

Modeling heap memory

```
int main ( ) {
  read(x);
  a = malloc(100);
  clear(a);
  a[x] = 5
  assert(a[x] != a[2*x - 2]);
}
```

```
int main ( ) {
  read(x);
  a = Ha;
  for (int i=0; i<100; i++) write(Ha, i, 0);
  write(Ha, x, 5);
  assert(read(Ha, x) != read(Ha, 2*x - 2));
}
```

Modeling heap memory

```
int main ( ) {
  read(x);
  a = malloc(100);
  clear(a);
  a[x] = 5
  assert(a[x] != a[2*x - 2]);
}
```

```
int main ( ) {
  read(x);
  a = Ha;
  for (int i=0; i<100; i++) write(Ha, i, 0);
  write(Ha, x, 5);
  assert(read(Ha, x) != read(Ha, 2*x - 2));
}
```

== *Use the array theory to model H* ==

Array theory

Axioms

- $\forall a\ \forall i\ \forall v\ (read(write(a, i, v), i) = v)$
- $\forall a\ \forall i\ \forall j\ \forall v\ (i \neq j \rightarrow read(write(a, i, v), j) = read(a, j))$
- $\forall a\ \forall b\ ((\forall i\ (read(a, i) = read(b, i))) \rightarrow a = b$

Virtualization

S2E[2] enables symbolic execution for binaries running in a virtualized environment (QEMU), enabling whole system verification.

[2] https://s2e.systems

Applications of Symbolic Execution

- Test-case generation (path coverage)

Applications of Symbolic Execution

- Test-case generation (path coverage)
- Program debugging [Chandra et al., ICSE 2011]

Applications of Symbolic Execution

- Test-case generation (path coverage)
- Program debugging [Chandra et al., ICSE 2011]
- Program repair [Nguyen et al., ICSE 2013; Mechtaev et al. ICSE 2016]

Applications of Symbolic Execution

- Test-case generation (path coverage)
- Program debugging [Chandra et al., ICSE 2011]
- Program repair [Nguyen et al., ICSE 2013; Mechtaev et al. ICSE 2016]
- Bucketing tests [Pham et al., FASE 2017]
- Debugging [Verma et al. FSE 2017, Verma et al. CGO 2020]
- Synthesis [Pandey et al. FSE 2018]

Angelic Degugging [Chandra et al. ICSE '11]

Objective

Given a informal specification as a set of tests, can we identify which expressions are ikely to be buggy?

Angelic Degugging [Chandra et al. ICSE '11]

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (t3 && t2) max = z;
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

Angelic Degugging [Chandra et al. ICSE '11]

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (t3 && t2) max = z;
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

What is the bug?

Angelic Degugging [Chandra et al. ICSE '11]

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (t3 && t2) max = z;
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

What is the bug?

How to fix the bug?

Angelic Degugging [Chandra et al. ICSE '11]

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (t3 && t2) max = z; // bug
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

Angelic Degugging [Chandra et al. ICSE '11]

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (t3 && t2) max = z; // bug
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

Test	Input	Output	Status
I1	8, 2, 4	8	Pass
I2	1, 2, 4	0	Fail
I3	7, 5, 4	7	Pass
I4	2, 5, 1	5	Pass

Approach

Define scope of debugging

E = All expressions (and subexpressions) within a user-specified scope.
Expressions at distinct program locations are different expressions.

Approach

Define scope of debugging

E = All expressions (and subexpressions) within a user-specified scope.
Expressions at distinct program locations are different expressions.

Test for *angelic* values on *failing* tests

$\forall e \in E : AngelicTest(P, I, e) = \exists \alpha . Test(P[\alpha/e], I)$

Approach

Define scope of debugging

E = All expressions (and subexpressions) within a user-specified scope.
Expressions at distinct program locations are different expressions.

Test for *angelic* values on *failing* tests

$\forall e \in E : AngelicTest(P, I, e) = \exists \alpha . Test(P[\alpha/e], I)$

Check for regressions on *passing* tests

$\forall e \in E : FlexTest(P, I, e) = \exists \alpha . (Test(P[\alpha/e], I) \wedge \alpha \neq Eval(P, I, e)$

Approach

Define scope of debugging

E = All expressions (and subexpressions) within a user-specified scope.
Expressions at distinct program locations are different expressions.

Test for *angelic* values on *failing* tests

$\forall e \in E : AngelicTest(P, I, e) = \exists \alpha.Test(P[\alpha/e], I)$

Check for regressions on *passing* tests

$\forall e \in E : FlexTest(P, I, e) = \exists \alpha.(Test(P[\alpha/e], I) \wedge \alpha \neq Eval(P, I, e)$

Collect suspicious expressions

$\{e \mid e \in E \wedge AngelicTest(P, I_f, e) \wedge \forall i \in [i..k].FlexTest(P, I_{p_i}, e)\}$

Angelic non-determinism

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x)

  if (*) max = z; // angelic
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

Angelic non-determinism

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x)

  if (*) max = z; // angelic
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

- Say, E = {t1, t2, t3, !t1, !t2, !t3, t1 && !t3, t2 && !t1, t3 && t2}
- For e = (t3 && t2), value 1 passes test
- There exist alternate values for e (i.e. for (t3 && t2)), different than the original values, that still passes all passing tests
- So, e = (t3 && t2) is identified as a suspicious expression

Understanding FlexTest

Ideal check

Given a suspicious expression e and a test I, does Test(P[e'/e], I) hold for an alternate expression e'?

Understanding FlexTest

Ideal check

Given a suspicious expression e and a test I, does Test(P[e'/e], I) hold for an alternate expression e'?

Requires us to know the *repaired* expression e'!

Understanding FlexTest

Ideal check

Given a suspicious expression e and a test I, does Test(P[e'/e], I) hold for an alternate expression e'?

Requires us to know the *repaired* expression e'!

Approximate check

Given a suspicious expression e and a test I, does Test(P[w'/e], I), hold for an alternate *value* w', $w \neq w'$, where w is the value for the expression e when P is run on I?

Understanding FlexTest

Ideal check

Given a suspicious expression e and a test I, does Test(P[e'/e], I) hold for an alternate expression e'?

Requires us to know the *repaired* expression e'!

Approximate check

Given a suspicious expression e and a test I, does Test(P[w'/e], I), hold for an alternate *value* w', $w \neq w'$, where w is the value for the expression e when P is run on I?

Do we lose anything?

Approximate Check

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;
  if (t3 && t2) max = z; // bug

  output(max);
}
```

Approximate Check

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;
  if (t3 && t2) max = z; // bug

  output(max);
}
```

- A non-zero value of expression (t3 && t2) breaks other passing tests, violating FlexText!

Approximate Check

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;
  if (t3 && t2) max = z; // bug

  output(max);
}
```

- A non-zero value of expression (t3 && t2) breaks other passing tests, violating FlexText!
- So, we may lose on some suspicious expressions.

Approximate Check

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;
  if (t3 && t2) max = z; // bug

  output(max);
}
```

- A non-zero value of expression (t3 && t2) breaks other passing tests, violating FlexText!
- So, we may lose on some suspicious expressions.
- But in this case, there is another repair expression, t2 at the last condition[a], that passes $FlexTest$

[a] recall that all expressions at distinct lines are different

How to realize angelic non-determinism using Symbolic Execution: AngelicTest and FlexTest

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x)

  if (a = symbolic()) //(t3 && t2)
        max = z;
  assume(a != (t3 && t2))
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
  assume(max == expected_max);
}
```

How to realize angelic non-determinism using Symbolic Execution: AngelicTest and FlexTest

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x)

  if (a = symbolic()) //(t3 && t2)
        max = z;
  assume(a != (t3 && t2))
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
  assume(max == expected_max);
}
```

- AngelicTest: Run program on concrete inputs, fresh symbolic variable for candidate expression, output assumed to be expected output.
- FlexTest: Run program on concrete inputs, fresh symbolic variable for candidates, *assume* that the symbolic variable takes a different value than actual expression, output assumed to be expected output.

Discussion

- Only works on 1-fixable programs
- Evaluated on JTOPAS, an open-source Java library for parsing arbitrary text, with 10 seeded faults; could identify 4 of them.

What about repair? (SemFix, ICSE 2013)

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  //(t3 && t2)
  if f(x, y, z, t1, t2, t2)
        max = z;
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

What about repair? (SemFix, ICSE 2013)

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  //(t3 && t2)
  if f(x, y, z, t1, t2, t2)
        max = z;
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

What about repair? (SemFix, ICSE 2013)

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  //(t3 && t2)
  if f(x, y, z, t1, t2, t2)
        max = z;
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

- Instead of non-determinism, maintain an uninterpreted function to "capture" the semantics of the correct expression;

What about repair? (SemFix, ICSE 2013)

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  //(t3 && t2)
  if f(x, y, z, t1, t2, t2)
        max = z;
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

- Instead of non-determinism, maintain an uninterpreted function to “capture” the semantics of the correct expression;
- Run the program on all inputs to collect “enough” examples for the semantics of the correct expression;

What about repair? (SemFix, ICSE 2013)

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  //(t3 && t2)
  if f(x, y, z, t1, t2, t2)
        max = z;
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

- Instead of non-determinism, maintain an uninterpreted function to "capture" the semantics of the correct expression;
- Run the program on all inputs to collect "enough" examples for the semantics of the correct expression;
- *Synthesize* the correct expression according to the semantics.

SemFix

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (a = symbolic()) max = z; //
     bug
  assume(a != (t3 && t2))
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

SemFix

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (a = symbolic()) max = z; //
    bug
  assume(a != (t3 && t2))
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

Test	f(x,y,z,t1,t2,t3)
I1	$f(8,2,4,1,0,0) = 0$
I2	$f(1,2,4,0,0,1) = 1$
I3	$f(7,5,4,1,1,0) = 0$
I4	$f(2,5,1,0,1,0) = 0$

SemFix

```
int main ( ) {
  read(x, y, z);
  t1 = (x >= y);
  t2 = (y >= z);
  t3 = (z >= x);

  if (a = symbolic()) max = z; //
     bug
  assume(a != (t3 && t2))
  if (t1 && !t3) max = x;
  if (t2 && !t1) max = y;

  output(max);
}
```

Test	f(x,y,z,t1,t2,t3)
I1	f(8,2,4,1,0,0) = 0
I2	f(1,2,4,0,0,1) = 1
I3	f(7,5,4,1,1,0) = 0
I4	f(2,5,1,0,1,0) = 0

(t3 && !t2) synthesized!

Angelix [ICSE '16]

- What about multi-line repairs?

Angelix [ICSE '16]

- What about multi-line repairs?
- Challenge: The value from one repaired expression may be required to feed into the repair of another expression:

Angelix [ICSE '16]

- What about multi-line repairs?
- Challenge: The value from one repaired expression may be required to feed into the repair of another expression:
  - The dependencies of which repairs feed into which, can be represented as a forest — angelic forest;

Angelix [ICSE '16]

- What about multi-line repairs?
- Challenge: The value from one repaired expression may be required to feed into the repair of another expression:
  - The dependencies of which repairs feed into which, can be represented as a forest — angelic forest;
  - The synthesis of repair expressions is done on this angelic forest;

Angelix [ICSE '16]

- What about multi-line repairs?
- Challenge: The value from one repaired expression may be required to feed into the repair of another expression:
  - The dependencies of which repairs feed into which, can be represented as a forest — angelic forest;
  - The synthesis of repair expressions is done on this angelic forest;
  - The angelic forest is independent of the size of the program, and only depends on the domain of candidate repair expressions.

Angelix [ICSE '16]

- What about multi-line repairs?
- Challenge: The value from one repaired expression may be required to feed into the repair of another expression:
  - The dependencies of which repairs feed into which, can be represented as a forest — angelic forest;
  - The synthesis of repair expressions is done on this angelic forest;
  - The angelic forest is independent of the size of the program, and only depends on the domain of candidate repair expressions.
- The synthesis procedure can be seen as synthesizing higher-order functions (symbolic execution and the angelic forest, however, allow you to skip this complexity and allows synthesis with values only)

Concluding Remarks

- In terms of industrial adoption, symbolic execution is perhaps one of the most successful outcomes from PL and SE research.
- Engines like SAGE and KLEE are quite mature, and are being used routinely in industry and academia.

Acknowledgements: Some of the slides are from the ISSTA 2019 talk of my student, Awanish Pandey.