Introduction to Formal Methods

Lecture 12

Symbolic and Concolic Execution

Hossein Hojjat & Fatemeh Ghassemi

November 4, 2018

Testing

- In practice, most common form of bug-detection
- Each test explores only one possible execution of the system

```
assert(f(2) == 7);
```

- We *hope* test cases generalize, but no guarantees
- Symbolic execution generalizes testing
- Allows *unknown* symbolic variables in evaluation

```
x = α; assert(f(x) == 3*x + 1);
```

Symbolic Execution

- Originally proposed by James King (among others)

  "Symbolic execution and program testing",
  Communications of the ACM, 1976.
- Recent advances in SMT solvers:
  renewed interest in symbolic execution!
- Companies like Microsoft use tools based on symbolic execution to find serious errors and security vulnerabilities
- Some tools based on symbolic execution: Symbolic PathFinder (SPF), KLEE, DART, CUTE, CREST, S2E, ...

Symbolic Execution: idea

```
void f (x, y) {
  if (x > y) {
    x = x + y;
    y = x − y;
    x = x − y;
    if (x − y > 0)
      assert false
  }
}
```

- Execute the program on symbolic values

Symbolic Execution: idea

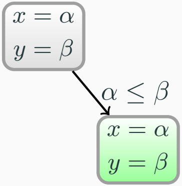
```
void f (x, y) {
  if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
      assert false
  }
}
```

$$x = \alpha$$
$$y = \beta$$

- Execute the program on symbolic values
- Symbolic state maps variables to symbolic values

Symbolic Execution: idea

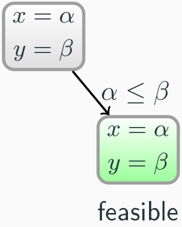
```
void f (x, y) {
  if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
      assert false
  }
}
```



- Execute the program on symbolic values
- Symbolic state maps variables to symbolic values
- Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far

Symbolic Execution: idea

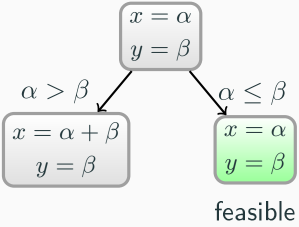
```
void f (x, y) {
  if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
      assert false
  }
}
```



- Execute the program on symbolic values
- Symbolic state maps variables to symbolic values
- Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far
- All paths in the program form its execution tree, in which some paths are *feasible* and some are *infeasible*

Symbolic Execution: idea

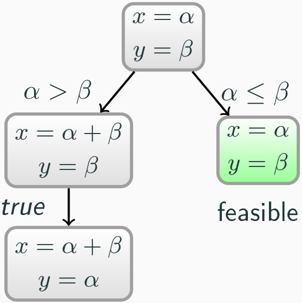
```
void f (x, y) {
  if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
      assert false
  }
}
```



- Execute the program on symbolic values
- Symbolic state maps variables to symbolic values
- Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far
- All paths in the program form its execution tree, in which some paths are *feasible* and some are *infeasible*

Symbolic Execution: idea

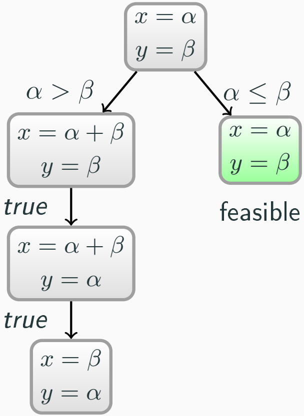
```
void f (x, y) {
  if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
      assert false
  }
}
```



- Execute the program on symbolic values
- Symbolic state maps variables to symbolic values
- Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far
- All paths in the program form its execution tree, in which some paths are *feasible* and some are *infeasible*

Symbolic Execution: idea

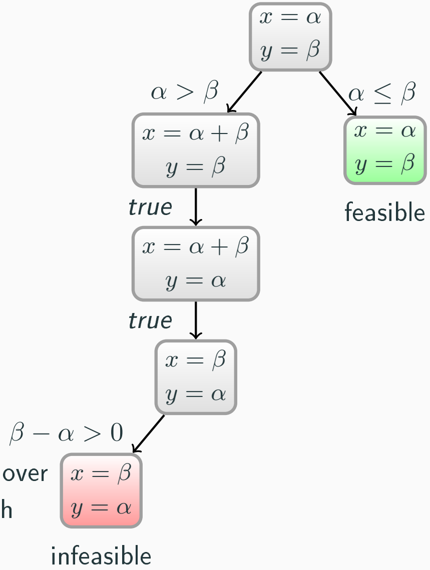
```
void f (x, y) {
  if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
      assert false
  }
}
```



- Execute the program on symbolic values
- Symbolic state maps variables to symbolic values
- Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far
- All paths in the program form its execution tree, in which some paths are *feasible* and some are *infeasible*

Symbolic Execution: idea

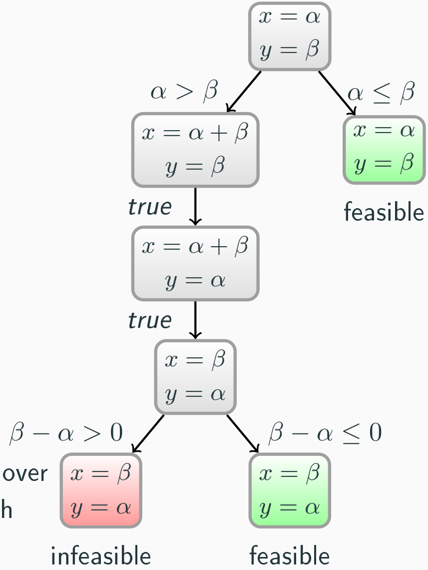
```
void f (x, y) {
  if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
      assert false
  }
}
```



- Execute the program on symbolic values
- Symbolic state maps variables to symbolic values
- Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far
- All paths in the program form its execution tree, in which some paths are *feasible* and some are *infeasible*

Symbolic Execution: idea

```
void f (x, y) {
  if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
      assert false
  }
}
```



- Execute the program on symbolic values
- Symbolic state maps variables to symbolic values
- Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far
- All paths in the program form its execution tree, in which some paths are *feasible* and some are *infeasible*

Symbolic Representation of State

- Let the set of variables be $V = \{x_1, \cdots, x_n\}$
- Program state in symbolic execution

$$\exists \alpha_1 \cdots \alpha_n . \underbrace{(x_1 = e_1) \land \cdots \land (x_n = e_n)}_{\sigma_s} \land P_C$$

(think of α_i as symbolically representing the initial value of x_i)

- e_i term describing symbolic state of the variables x_i
- P_C path condition
- Variables $x_1, \cdots, x_n$ do not appear in $e_1, \cdots, e_n, P_C$
- $\alpha_1, \cdots, \alpha_n$ fresh variables that do not appear in the program

Symbolic Execution and Strongest Postconditions

- Strongest postcondition introduces no new existential quantifiers for this form of symbolic representation

$$sp\Big(\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_n = e_n)\wedge P_C, x_i := E\Big)$$

Symbolic Execution and Strongest Postconditions

- Strongest postcondition introduces no new existential quantifiers for this form of symbolic representation

$$
\begin{aligned}
&\mathit{sp}\Big(\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_n = e_n)\wedge P_C, x_i := E\Big)\\
&= \exists\beta.\Big(\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_n = e_n)\wedge P_C\Big)[x_i := \beta]\wedge(x_i = E[x_i := \beta])
\end{aligned}
$$

Symbolic Execution and Strongest Postconditions

- Strongest postcondition introduces no new existential quantifiers for this form of symbolic representation

$$\mathit{sp}\Big(\exists\alpha_1\cdots\alpha_n.(x_1=e_1)\wedge\cdots\wedge(x_n=e_n)\wedge P_C, x_i:=E\Big)$$
$$=\exists\beta.\Big(\exists\alpha_1\cdots\alpha_n.(x_1=e_1)\wedge\cdots\wedge(x_n=e_n)\wedge P_C\Big)[x_i:=\beta]\wedge(x_i=E[x_i:=\beta])$$
$$=\exists\beta.\ \ \exists\alpha_1\cdots\alpha_n.(x_1=e_1)\wedge\cdots\wedge(\beta=e_i)\wedge\cdots\wedge(x_n=e_n)\wedge P_C\wedge(x_i=E[x_i:=\beta])$$

$(x_i$ not in $e_1,\cdots,e_n,P_C)$

Symbolic Execution and Strongest Postconditions

- Strongest postcondition introduces no new existential quantifiers for this form of symbolic representation

$$
\begin{aligned}
&\mathit{sp}\Big(\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_n = e_n)\wedge P_C, x_i := E\Big)\\
= \exists\beta.&\Big(\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_n = e_n)\wedge P_C\Big)[x_i := \beta]\wedge(x_i = E[x_i := \beta])\\
= \exists\beta.&\ \ \exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(\beta = e_i)\wedge\cdots\wedge(x_n = e_n)\wedge P_C\wedge(x_i = E[x_i := \beta])\\
&\quad (x_i \text{ not in } e_1,\cdots,e_n,P_C)\\
= \quad&\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_i = E[x_i := e_i])\wedge\cdots\wedge(x_n = e_n)\wedge P_C\\
&\quad (\text{one-point rule, move equation for } x_i \text{ inside})
\end{aligned}
$$

Symbolic Execution and Strongest Postconditions

- Strongest postcondition introduces no new existential quantifiers for this form of symbolic representation

$$
\begin{aligned}
& \mathit{sp}\Big(\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_n = e_n)\wedge P_C, x_i := E\Big) \\
= \exists\beta.&\Big(\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_n = e_n)\wedge P_C\Big)[x_i := \beta]\wedge(x_i = E[x_i := \beta]) \\
= \exists\beta.&\ \ \exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(\beta = e_i)\wedge\cdots\wedge(x_n = e_n)\wedge P_C\wedge(x_i = E[x_i := \beta]) \\
& \quad (x_i \text{ not in } e_1,\cdots,e_n,P_C) \\
= \quad & \exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_i = E[x_i := e_i])\wedge\cdots\wedge(x_n = e_n)\wedge P_C \\
& \quad (\text{one-point rule, move equation for } x_i \text{ inside}) \\
= \quad & \exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_i = E[x_1,...,x_n := e_1,...,e_n])\wedge\cdots\wedge(x_n = e_n)\wedge P_C \\
& \quad (\text{replace } x_1,...,x_n \text{ vars in } E)
\end{aligned}
$$

Symbolic Execution and Strongest Postconditions

- Strongest postcondition introduces no new existential quantifiers for this form of symbolic representation

$$
\begin{aligned}
&\mathit{sp}\Big(\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_n = e_n)\wedge P_C, x_i := E\Big)\\
= \exists\beta.&\Big(\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_n = e_n)\wedge P_C\Big)[x_i := \beta]\wedge(x_i = E[x_i := \beta])\\
= \exists\beta.&\ \ \exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(\beta = e_i)\wedge\cdots\wedge(x_n = e_n)\wedge P_C\wedge(x_i = E[x_i := \beta])\\
&\quad (x_i \text{ not in } e_1,\cdots,e_n,P_C)\\
= \quad&\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_i = E[x_i := e_i])\wedge\cdots\wedge(x_n = e_n)\wedge P_C\\
&\quad (\text{one-point rule, move equation for } x_i \text{ inside})\\
= \quad&\exists\alpha_1\cdots\alpha_n.(x_1 = e_1)\wedge\cdots\wedge(x_i = E[x_1,...,x_n := e_1,...,e_n])\wedge\cdots\wedge(x_n = e_n)\wedge P_C\\
&\quad (\text{replace } x_1,...,x_n \text{ vars in } E)
\end{aligned}
$$

- Calculating *sp* of $x_i := E$ consists of

1. Evaluate E in the current state (i.e. $E[x_1, ..., x_n := e_1, ..., e_n]$)
2. Update the equation for x_i to the new E

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- Can you find the inputs that make the program reach the ERROR?
- Let's execute this example with classic symbolic execution

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- `read()` reads a value from input
- We don't know what those read values are so we set them to fresh symbolic values α and β
- P_C is true because so far we have not executed any conditionals

$\sigma_s : x = \alpha \wedge y = \beta \qquad P_C : \textit{true}$

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta$ $\quad P_C$: *true*

- We simply execute the function `twice()` and add the new symbolic value for `z`

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- We fork the analysis into 2 paths: the true and the false path
- We need to duplicate the state of the analysis

This is the result if $x = z$:

$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta \qquad P_C : \alpha = 2\beta$

This is the result if $x \neq z$:

$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta \qquad P_C : \alpha \neq 2\beta$

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- We can avoid further exploring a path if we know the constraint P_C is unsat
- In this example, both P_C's are sat so we need to keep exploring both paths

This is the result if $x = z$:

$$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta \qquad P_C : \alpha = 2\beta$$

This is the result if $x \neq z$:

$$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta \qquad P_C : \alpha \neq 2\beta$$

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- Let's explore the path when `x == z` is true
- Once again we get 2 more paths

This is the result if $x > y + 10$:

$$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta$$
$$P_C : \alpha = 2\beta \wedge \alpha > \beta + 10$$

This is the result if $x \leq y + 10$:

$$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta$$
$$P_C : \alpha \neq 2\beta \wedge \alpha \leq \beta + 10$$

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- So the following path reaches ERROR

This is the result if $x > y + 10$:

$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta$

$P_C : \alpha = 2\beta \wedge \alpha > \beta + 10$

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- So the following path reaches ERROR

This is the result if $x > y + 10$:

$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta$

$P_C : \alpha = 2\beta \wedge \alpha > \beta + 10$

- We can now ask the SMT solver for a satisfying assignment to the P_C formula
- For instance, $\alpha = 40$, $\beta = 20$ is a satisfying assignment.
- Running the program with those concrete inputs triggers the error

Handling Loops: a limitation

```c
int F(unsigned int k) {
  int sum = 0;
  int i = 0;
  for ( ; i < k; i++)
    sum += i;
  return sum;
}
```

- A serious limitation of symbolic execution is unbounded loops
- Symbolic execution runs the program for a finite number of paths
- What if we do not know the bound on a loop?

Loops and Recursion

Dealing with infinite execution trees:

- Finitize paths by limiting the size of P_C (bounded verification)
- Use loop invariants (verification)

```
while (b) {
  c
}
assert(P)
```

```
assert(I)
havoc(x1,...,xn)
assume(I)
if (b) {
  c
  assert(I)
} else
  assert(P)
```

where `x1,···,xn` are variables modified in `c` and `I` is the loop invariant

When Constraint Solving Fails

- Despite best efforts, the program may be using constraints in a fragment which the SMT solver does not handle well
- For instance, suppose the SMT solver does not handle non-linear constraints well
  - Decision problem for non-linear integer arithmetic is undecidable
  - We can encode the Halting problem for Turing machines in non-linear integer arithmetic
- Let us consider a modification of our running example

Modified Example

```
int twice(int v) {
  return v * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- Here, we changed the `twice()` function to contain a non-linear result
- Let's see what happens when we symbolically execute the program now

Modified Example

```
int twice(int v) {
  return v * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

This is the result if $x = z$:

$$\sigma_s : x = \alpha \wedge y = \beta \wedge z = \beta \times \beta$$

$$P_C : \alpha = \beta \times \beta$$

- Now, if we are to invoke the SMT solver with the P_C formula, it would be unable to compute satisfying assignments
- We cannot know whether the path is feasible or not

Solution: Concolic Execution

- Concolic Execution: combines both **symbolic** execution and **concrete** (normal) execution
- The basic idea is to have the concrete execution drive the symbolic execution
- Here, the program runs as usual (it needs to be given some input), but in addition it also maintains the usual symbolic information

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- `read()` reads a value from input
- Suppose we read `x = 22` and `y = 7`
- We will keep both the concrete store and the symbolic store and path constraint

$$\sigma : x = 22 \wedge y = 7$$
$$\sigma_s : x = \alpha \wedge y = \beta$$
$$P_C : \textit{true}$$

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

$$\sigma : x = 22 \land y = 7 \land z = 14$$
$$\sigma_s : x = \alpha \land y = \beta \land z = 2\beta$$
$$P_C : \textit{true}$$

- The concrete execution will now take the `else` branch of `x == z`

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- Hence, we get:

$$\sigma : x = 22 \wedge y = 7 \wedge z = 14$$

$$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta$$

$$P_C : \alpha \neq 2\beta$$

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- At this point, concolic execution decides that it would like to explore the *true* branch of `x == z` and hence it needs to generate concrete inputs to explore it
- It negates the P_C constraint, obtaining:

$$P_C: \ \alpha = 2\beta$$

- It then calls the SMT solver to find a satisfying assignment of that constraint
- Let us suppose the SMT solver returns:

$$\alpha = 2, \beta = 1$$

- The concolic execution then runs the program with this input

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- With the input $x = 2$, $y = 1$ we reach this program point with the following information:

$$\sigma : x = 2 \wedge y = 1 \wedge z = 2$$
$$\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta$$
$$P_C : \alpha = 2\beta$$

- Continuing further we get:

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- We reach the `else` branch of `x>y+10`

  $\sigma : x = 2 \wedge y = 1 \wedge z = 2$

  $\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta$

  $P_C : \alpha = 2\beta \wedge \alpha \leq \beta + 10$

- Again, concolic execution may want to explore the `true` branch of `x>y+10`

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- We reach the else branch of `x>y+10`

  $\sigma : x = 2 \wedge y = 1 \wedge z = 2$

  $\sigma_s : x = \alpha \wedge y = \beta \wedge z = 2\beta$

  $P_C : \alpha = 2\beta \wedge \alpha \leq \beta + 10$

- Concolic execution now negates the conjunct $\alpha \leq \beta + 10$ obtaining: $\alpha = 2\beta \wedge \alpha > \beta + 10$
- A satisfying assignment is: $\alpha = 30 \wedge \beta = 15$

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- If we run the program with the input:

$$x = 30 \land y = 15$$

- We will now reach the ERROR state
- As we can see from this example, by keeping the symbolic information, the concrete execution can use that information in order to obtain new inputs.

Non-linear constraints

Let us return to the problem of non-linear constraints

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- Let us again consider our example and see what concolic execution would do with non-linear constraints

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- `read()` reads a value from input
- Suppose we read `x = 22` and `y = 7`

$\sigma : x = 22 \wedge y = 7$

$\sigma_s : x = \alpha \wedge y = \beta$

P_C : *true*

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

$$\sigma : x = 22 \wedge y = 7 \wedge z = 49$$

$$\sigma_s : x = \alpha \wedge y = \beta \wedge z = \beta \times \beta$$

$$P_C : \textit{true}$$

- The concrete execution will now take the `else` branch of `x == z`

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- Hence, we get:

$$\sigma : x = 22 \wedge y = 7 \wedge z = 49$$

$$\sigma_s : x = \alpha \wedge y = \beta \wedge z = \beta \times \beta$$

$$P_C : \alpha \neq \beta \times \beta$$

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- We have a non-linear constraint $\alpha = \beta \times \beta$
- If we would like to explore the true branch we negate the constraint, obtaining $\alpha = \beta \times \beta$ but again we have a non-linear constraint
- In this case, concolic execution simplifies the constraint by plugging in the concrete values for β
- $\beta = 7$ so we obtain the simplified constraint: $\alpha = 49$
- Hence, it now runs the program with the input

$$x = 49 \ , \ y = 7$$

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}
void test(int x,int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}
int main() {
  x = read();
  y = read();
  test(x,y);
}
```

- Running with the input

$$x = 49 \ , \ y = 7$$

- We reach the error state.

Notice that with these inputs, if we try to simplify non-linear constraints by plugging in concrete values (as concolic execution does), then concolic execution we will never reach the `else` branch of the `if (x > y + 10)` statement

Reference

Cristian Cadar, Koushik Sen, “Symbolic execution for software testing: three decades later”, Communications of the ACM, Volume 56, Issue 2, February 2013, Pages 82-90.