Symbolic Model Checking

EECS 4315

www.eecs.yorku.ca/course/4315/

Model checking

Question

Which major problem does model checking face?

Model checking

Question

Which major problem does model checking face?

Answer

The state space explosion problem: the state space is too large to be analyzed as the model checker runs out of memory or time.

Model checking

Question

Which major problem does model checking face?

Answer

The state space explosion problem: the state space is too large to be analyzed as the model checker runs out of memory or time.

One approach

Combining multiple transitions into a single one.

Model checking

Question

Which major problem does model checking face?

Answer

The state space explosion problem: the state space is too large to be analyzed as the model checker runs out of memory or time.

Another approach

Representing the model in a symbolic way where sets of states and sets of transitions are represented rather than single states and single transitions.

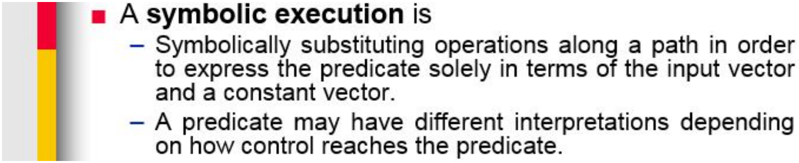
Symbolic execution

James King. Symbolic execution and program testing, *Communications of the ACM*, 19(7): 385–394, July 1976.

“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.”

“The symbolic execution of IF statements requires theorem proving which, even for modest programming languages, is mechanically impossible.”

Symbolic execution

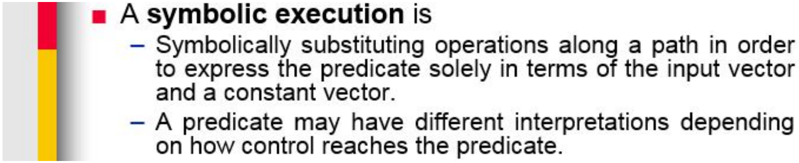


- A **symbolic execution** is
  - Symbolically substituting operations along a path in order to express the predicate solely in terms of the input vector and a constant vector.
  - A predicate may have different interpretations depending on how control reaches the predicate.

Question

Have you seen this before? If so, where?

Symbolic execution



- A **symbolic execution** is
  - Symbolically substituting operations along a path in order to express the predicate solely in terms of the input vector and a constant vector.
  - A predicate may have different interpretations depending on how control reaches the predicate.

Question

Have you seen this before? If so, where?

Answer

In EECS 4313 Software Engineering Testing.

Branch selection

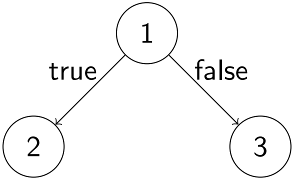
```
package example;

public class Branch {
  public void select(int x, int y) {
    if (x < 0) {
      x = -x;
    }
    if (y < 0) {
      y = -y;
    }
    if (x < y) {
      System.out.println("abs(x) < abs(y)");
    } else if (x == 0) {
      System.out.println("x == y == 0");
    } else {
      System.out.println("x >= y >= 0");
    }
  }
}
```

Path condition

```
if (x < 0) {
  x = -x;
}
```

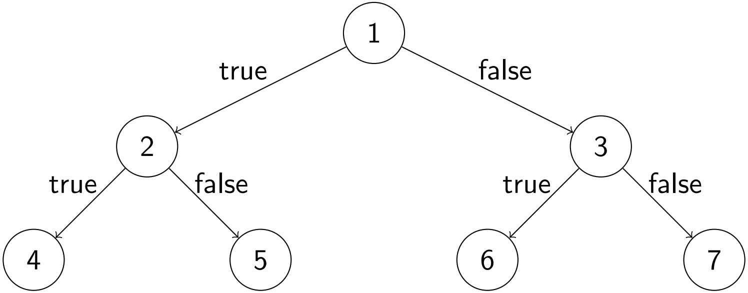
state	path condition
1	true
2	$x < 0$
3	$\neg(x < 0)$



Path condition

```
if (x < 0) {
  x = -x;
}
if (y < 0) {
  y = -y;
}
```

state	path condition
4	$x < 0 \wedge y < 0$
5	$x < 0 \wedge \neg(y < 0)$
6	$\neg(x < 0) \wedge y < 0$
7	$\neg(x < 0) \wedge \neg(y < 0)$



Path condition

The path condition corresponding to one of the executions ending with

```
System.out.println("x >= y >= 0");
```

is

$$\neg(x < 0) \wedge \neg(y < 0) \wedge \neg(x < y) \wedge \neg(x = 0)$$

Path condition

The path condition corresponding to one of the executions ending with

```
System.out.println("x >= y >= 0");
```

is

$$\neg(x < 0) \wedge \neg(y < 0) \wedge \neg(x < y) \wedge \neg(x = 0)$$

Theorem provers such as Z3 and CVC4 can find an assignment to the values x and y that satisfies the above path condition. For example, $x = 2147483647$ and $y = 0$.

Path condition

The path condition corresponding to one of the executions ending with

```
System.out.println("x >= y >= 0");
```

is

$$\neg(x < 0) \wedge \neg(y < 0) \wedge \neg(x < y) \wedge \neg(x = 0)$$

Theorem provers such as Z3 and CVC4 can find an assignment to the values x and y that satisfies the above path condition. For example, $x = 2147483647$ and $y = 0$.

This results in

```
@Test
public void test() {
  Branch branch = new Branch();
  branch.select(2147483647, 0);
}
```

jpf-symbc

The JPF extension jpf-symbc combines symbolic execution with model checking. It generates test cases.

Corina Pasareanu, Peter Mehlitz, David Bushnell ,Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 15-26, Seattle, WA, USA, July 2008. ACM.

Debug option

Turn the debug option on by compiling the code with the `-g` option.

```
javac -g example/Branch.java
```

Application

Write an app that uses the methods to be tested.

```
public class Main {
  public static void main(String[] args) {
    Branch branch = new Branch();
    branch.select(1, 2);
  }
}
```

Application properties file

To execute a method symbolically, the user needs to specify in the application properties file which method arguments are concrete (`con`) and which are symbolic (`sym`).

For example, executing the method `select` of the class `Branch` of the package `example` with both argument concrete is specified in the following application properties file.

```
@using=jpf-symbc
target=Main
classpath=.
symbolic.method=example.Branch.select(con#con)
```

JPF's output

```
abs(x) < abs(y)
```

Application properties file

To execute a method symbolically, the user needs to specify in the application properties file which method arguments are concrete (`con`) and which are symbolic (`sym`).

For example, executing the method `select` of the class `Branch` of the package `example` with the first argument symbolic and the second argument concrete is specified in the following application properties file.

```
@using=jpf-symbc
target=Main
classpath=.
symbolic.method=example.Branch.select(sym#con)
```

JPF's output

```
abs(x) < abs(y)
x >= y >= 0
abs(x) < abs(y)
x >= y >= 0
```

Application properties file

To execute a method symbolically, the user needs to specify in the application properties file which method arguments are concrete (`con`) and which are symbolic (`sym`).

For example, executing the method `select` of the class `Branch` of the package `example` with the first argument concrete and the second argument symbolic is specified in the following application properties file.

```
@using=jpf-symbc
target=Main
classpath=.
symbolic.method=example.Branch.select(con#sym)
```

JPF's output

```
abs(x) < abs(y)
x >= y >= 0
abs(x) < abs(y)
x >= y >= 0
```

Application properties file

To execute a method symbolically, the user needs to specify in the application properties file which method arguments are concrete (`con`) and which are symbolic (`sym`).

For example, executing the method `select` of the class `Branch` of the package `example` with both argument symbolic is specified in the following application properties file.

```
@using=jpf-symbc
target=Main
classpath=.
symbolic.method=example.Branch.select(sym#sym)
```

JPF's output

```
abs(x) < abs(y)
x >= y >= 0
abs(x) < abs(y)
abs(x) < abs(y)
x >= y >= 0
abs(x) < abs(y)
x == y == 0
x >= y >= 0
```

Path conditions

```
import gov.nasa.jpf.symbc.Debug;

public class Main {
  public static void main(String[] args) {
    Branch branch = new Branch();
    branch.select(1, 2);
    Debug.printPC("\nPath Condition: ");
  }
}
```

Application properties file

```
@using=jpf-symbc
target=Main
classpath=.
symbolic.method=example.Branch.select(sym#sym)
```

JPF's output

```
abs(x) < abs(y)

Path Condition: constraint # = 3
(CONST_0 - x_1_SYMINT) < (CONST_0 - y_2_SYMINT) &&
y_2_SYMINT < CONST_0 &&
x_1_SYMINT < CONST_0
x >= y >= 0

Path Condition: constraint # = 3
(CONST_0 - x_1_SYMINT) >= (CONST_0 - y_2_SYMINT) &&
y_2_SYMINT < CONST_0 &&
x_1_SYMINT < CONST_0
abs(x) < abs(y)


...
```

Application properties file

```
@using=jpf-symbc
target=Main
classpath=.
symbolic.method=example.Branch.select(sym#sym)
listener=gov.nasa.jpf.symbc.SymbolicListener
```

JPF's output

```
abs(x) < abs(y)

Path Condition: constraint # = 3
(CONST_0 - x_1_SYMINT[2147483647]) < (CONST_0 - y_2_SYMINT
y_2_SYMINT[2147483647] < CONST_0 &&
x_1_SYMINT[2147483647] < CONST_0
x >= y >= 0

Path Condition: constraint # = 3
(CONST_0 - x_1_SYMINT[-2147483648]) >= (CONST_0 - y_2_SYMI
y_2_SYMINT[-2147483648] < CONST_0 &&
x_1_SYMINT[-2147483648] < CONST_0
abs(x) < abs(y)

...
```

Application

```
public class Main {
  public static void main(String[] args) {
    Branch branch = new Branch();
    branch.select(1, 2);
  }
}
```

Application properties file

```
@using=jpf-symbc
target=Main
classpath=.
symbolic.method=example.Branch.select(sym#sym)
listener=\
gov.nasa.jpf.symbc.sequences.SymbolicSequenceListener
```

JPF's output

```
==================================== Method Sequences
[select(2147483647,2147483647)]
[select(-2147483648,-2147483648)]
[select(-2147483648,0)]
[select(0,-2147483648)]
[select(0,0)]
[select(2147483647,0)]
```

JPF's output

```
================================== JUnit 4.0 test class
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class example_BranchTest {
  private example.Branch example_branch;

  @Before
  public void setUp() throws Exception {
    example_branch = new example.Branch();
  }

  @Test
  public void test0() {
    example_branch.select(2147483647,2147483647);
  }
```