Symbolic Execution

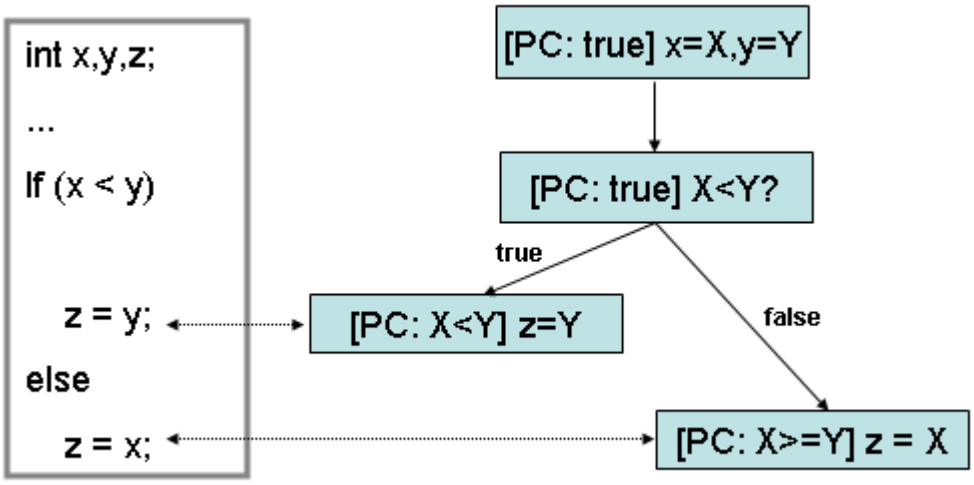

Figure 1: Symbolic Execution

This JPF extension performs symbolic execution of Java programs. One of the main applications is automated generation of test inputs that obtain high coverage (e.g. path coverage) of code. Other applications include error detection in concurrent programs that take inputs from unbounded domains and lightweight theorem proving.

The extension combines symbolic execution with model checking and constraint solving for test input generation. In this tool, programs are executed on symbolic inputs that cover all possible concrete inputs. Values of variables are represented as numeric expressions and constraints, generated from analysis of the code structure. These constraints are then solved to generate test inputs guaranteed to reach that part of code.

For example, in Figure 1 the path condition (**PC**) is initially **true**. If the program takes the `if` statement's `then` branch, the path condition will be **X<Y**. If the program takes the `if` statement's `else` branch, the path condition will be **X≥Y**.

For more information, please look at: http://ti.arc.nasa.gov/projects/advtesting/.

Implementation

At this time (December, 2007) symbolic execution works best for methods with integer arguments. We are now extending symbolic execution to work with double or float arguments so at the moment it does work in some cases, but not all.

Symbolic execution is implemented by a "non-standard" interpretation of bytecodes. The symbolic information is propagated via attributes associated with program variables, operands, etc. The current implementation uses Choco (pure Java, from sourceforge) for linear/non-linear integer/real constraints. Symbolic execution can start from any point in the program and it can perform mixed concrete/symbolic execution. State matching is turned off during symbolic execution.

Instructions

To execute a method symbolically, the user needs to specify which method arguments are symbolic/concrete. Globals (i.e. fields) can also be specified to be symbolic, via special annotations (to add more here...). Here is an example of a JPF run configuration. It enables symbolic execution of method "test" in main class "ExSymExe" (see the "test" directory), where first argument is symbolic, and second argument is concrete).

```
+vm.insn_factory.class=gov.nasa.jpf.symbc.SymbolicInstructionFactory
+jpf.listener=gov.nasa.jpf.symbc.SymbolicListener
+symbolic.method=test(sym#con)
+jpf.report.console.finished=
ExSymExe
```

A Simple but Complete Example

The following is a very simple example of doing symbolic execution with JPF. It assumes that you are using Eclipse, but the instructions should be easily adapted to other IDEs or the command line.

Suppose you have the following method in your class that you want to generate tests for:

```
public class MyClass1 {
    ... etc ...
    public int myMethod(int x, int y) {
        int z = x + y;
        if (z > 0) {
            z = 1;
        } else {
            z = z - x;
        }
        z = 2 * z;
        return z;
    }
    ... etc ...
}
```

You will need to create a driver that calls `myMethod()`. The driver could be a separate driver class or the `main()` method of one of your classes. For simplicity, we will write the driver as `MyClass1`'s `main()` method.

For this simple example, your driver just needs to call `myMethod()` with the right number and types of arguments and then print the path condition. Surprisingly, the exact arguments do not matter. Since you will be executing `myMethod()` symbolically, any concrete values you pass in will be replaced by symbolic values anyway.

At the moment the quickest way to see your test cases is to print out the path conditions by calling `gov.nasa.jpf.symbc.Debug.printPC()` in your driver. It's a bit primitive, but it is easy and we hope to have better facilities soon.

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

public class MyClass1 {

    ... etc ...
```

```
    // The method you need tests for
    public int myMethod(int x, int y) {
        int z = x + y;
        if (z > 0) {
            z = 1;
        } else {
            z = z - x;
        }
        z = 2 * z;
        return z;
    }

    ... etc ...

    // The test driver
    public static void main(String[] args) {
        MyClass1 mc = new MyClass1();
        int x = mc.myMethod(1, 2);
        Debug.printPC("\nMyClass1.myMethod Path Condition: ");
    }
}
```

So let's run this symbolically. The above code is in JPF's symbc examples directory, in the default package: `extensions/symbc/examples/MyClass1.java`.

In Eclipse, create a new run configuration for your JPF project:

- Select **Run → Run...**
- Click left on **Java Application**
- Click left on the "new launch configuration" button
- Enter a name for your run configuration
- Use `gov.nasa.jpf.JPF` as your main class

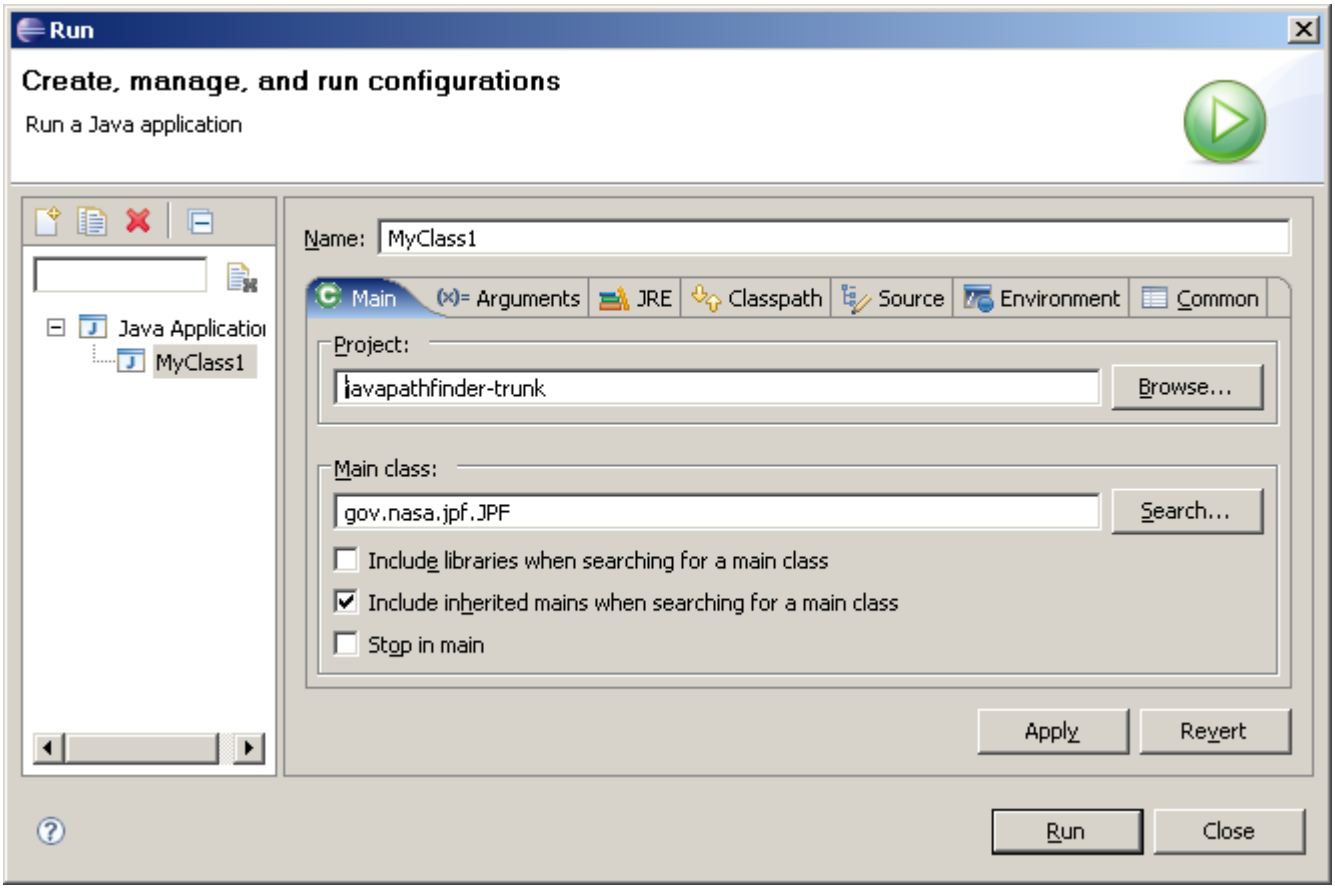

Figure 2: Eclipse Run Configuration for `MyClass1`, Main Tab

On the **Arguments** config page, paste the following into the **Program Arguments** box:

```
+vm.insn_factory.class=gov.nasa.jpf.symbc.SymbolicInstructionFactory
+vm.classpath=.
+vm.storage.class=
+symbolic.method=myMethod(sym#sym)
+jpf.report.console.finished=
MyClass1
```

For simple examples, most of these arguments will not change. The ones that will are `symbolic.method` and the class name (the last line). For larger examples, you may need to increase the memory, as shown here in the **VM Arguments** box.

The value for `symbolic.method` should be your method name followed by one "`sym`" for each of its arguments.

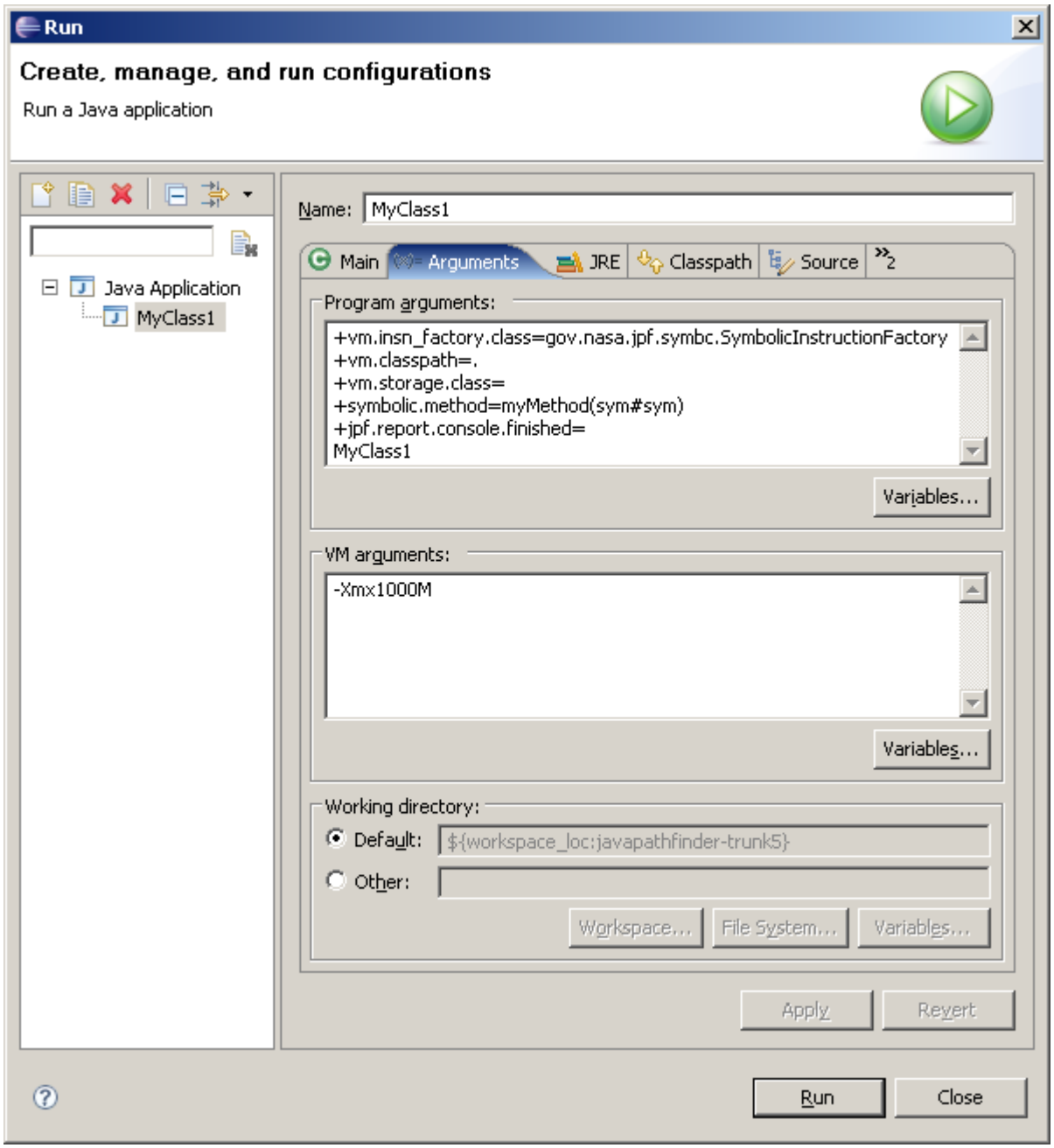

Figure 3: Eclipse Run Configuration for MyClass1, Program Arguments

And that's it for configuration. If you click on the **Run** button you will get something like this in the console window:

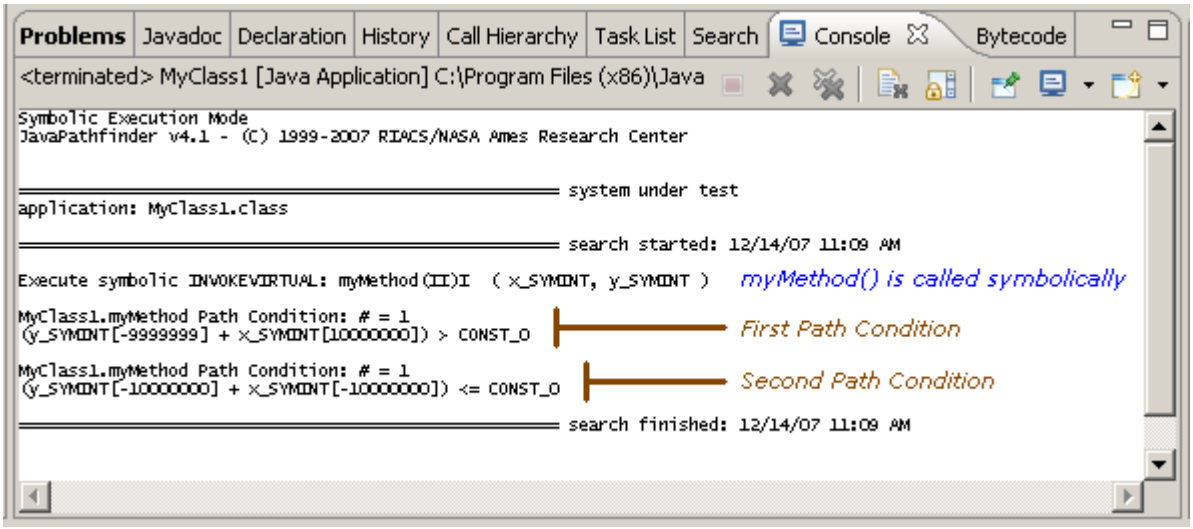

Figure 4: Eclipse Output for MyClass1

In the above output the path conditions (marked with brown arrows) show you the test cases:

```
Test Case 1:  y = -9999999,  x = 10000000
Test Case 2:  y = -10000000, x = 10000000
```

Test Case 1 corresponds to the `z>0` branch of `myMethod`'s `if` statement. Test Case 2 corresponds to the `z≤0` branch of the `if` statement.

Extending the Example: Filtering the Test Cases

Let's modify the `MyClass1.myMethod()` example a bit. Suppose you have another class with a slightly different method:

```
public class MyClass2 {
    ... etc ...
    private int myMethod2(int x, int y) {
        int z = x + y;
        if (z > 0) {
            z = 1;
        }
        if (x < 5) {
            z = -z;
        }
        return z;
    }
    ... etc ...
}
```

To generate test cases for this method you create a new Eclipse run configuration and add a `main()` method to `MyClass2`, as before:

```
+vm.insn_factory.class=gov.nasa.jpf.symbc.SymbolicInstructionFactory
+vm.classpath=.
+vm.storage.class=
+symbolic.method=myMethod2(sym#sym)
+jpf.report.console.finished=
```

```
MyClass2
```

The parts that have changed are in bold. Here is the code with the new driver:

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

public class MyClass2 {

    ... etc ...
    private int myMethod2(int x, int y) {
        int z = x + y;
        if (z > 0) {
            z = 1;
        }
        if (x < 5) {
            z = -z;
        }
        return z;
    }

    ... etc ...

    // The test driver
    public static void main(String[] args) {
        MyClass2 mc = new MyClass2();
        int x = mc.myMethod2(1, 2);
        Debug.printPC("\nMyClass2.myMethod2 Path Condition: ");
    }
}
```

If you run this as in the previous example, you will get four test cases which test the four possible paths through the method:

```
Test Case 1:  y = 10000000,  x = -9999999
Test Case 2:  y = -4,        x = 5
Test Case 3:  y = -10000000, x = -10000000
Test Case 4:  y = -10000000, x = 5
```

However, suppose that you are only interested in the test cases where exactly one of the `if` statements is executed. That is, you only care about Test Case 2 and Test Case 3. You could, of course, just run JPF as above and filter these out by hand. But that would be both tedious and inefficient if your `myMethod2()` had fifty consecutive `if` statements.

If you are allowed to annotate your method's code, there is another approach: use `Verify.ignoreIf()` to force JPF to backtrack when more than one `if` statement fires. For example, you can annotate `myMethod2()` as follows:

```
import gov.nasa.jpf.jvm.Verify;
import gov.nasa.jpf.symbc.Debug;
... etc ...

public class MyClass2 {

    ... etc ...
    private int myMethod2(int x, int y) {
        int jpfIfCounter = 0;
        int z = x + y;
```

```
        if (z > 0) {
            jpfIfCounter++;
            z = 1;
        }
        if (x < 5) {
            jpfIfCounter++;
            Verify.ignoreIf(jpfIfCounter > 1);
            z = -z;
        }
        Verify.ignoreIf(jpfIfCounter == 0);
        return z;
  }

    ... etc ...

    // The test driver
    public static void main(String[] args) {
        MyClass2 mc = new MyClass2();
        int x = mc.myMethod2(1, 2);
        Debug.printPC("\nMyClass2.myMethod2 Path Condition: ");
    }
}
```

The added annotations are shown in bold. This code is in `extensions/symbc/examples/MyClass2.java`.

Now you can execute this symbolically and you will only get the two test cases you need:

```
Test Case 1:  y = -4,        x = 5
Test Case 2:  y = -10000000, x = -10000000
```

Extending the Example: Adding Preconditions

Suppose again that you want to restrict the test cases generated, but now the problem is that you believe that your method will only be called with certain ranges of parameters. For example, in `MyClass1.myMethod()` you believe that `x` and `y` are restricted to the ranges $-100 \leq x \leq 100$ and $1 \leq y \leq 3$.

Enforcing these requirements during symbolic execution is easy — you do not even need to modify `myMethod()`'s original code. Instead you can use preconditions implemented in the driver.

For the sake of variety we will use a separate class to implement the driver, `MyDriver1`.

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

public class MyDriver1 {

    // The method whose parameters are marked as symbolic.
    private static void imposePreconditions(int x, int y) {
        MyClass1 mc = new MyClass1();

        // The preconditions
        if (-100 <= x  &&  x <= 100  &&  1 <= y  &&  y <= 3) {
            mc.myMethod(x, y);
            Debug.printPC("\nMyClass1.myMethod Path Condition: ");
        }
```

```
    }

    // The test driver
    public static void main(String[] args) {
        // Actual arguments are ignored when doing symbolic execution.
        imposePreconditions(1,2);
    }
}
```

Because you need preconditions and because you do not want to modify `MyClass1`, the method you call symbolically is `MyDriver1.imposePreconditions()`, not `MyClass1.myMethod()`. Note that `imposePreconditions()`'s parameters are `x` and `y`, the parameters to `myMethod()` that you want to be symbolic. This code is in `extensions/symbc/examples/MyDriver1.java`.

Create an Eclipse run configuration for `MyDriver1`, pasting the following into the **Program Arguments** box on the **Arguments** config page:

```
+vm.insn_factory.class=gov.nasa.jpf.symbc.SymbolicInstructionFactory
+vm.classpath=.
+vm.storage.class=
+symbolic.method=imposePreconditions(sym#sym)
+jpf.report.console.finished=
MyDriver1
```

Running `MyDriver1` produces the test cases with the desired range restrictions:

```
Test Case 1:  y = 1,  x = 0
Test Case 2:  y = 1,  x = -100
```

Floating Point Arguments

Methods with `float` and `double` parameters are handled exactly the same as ones with `int` parameters, though the output is slightly different. Here is a quick illustration.

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

public class MyClassFP {
    public double myMethodFP(double x, double y) {
        double z = x + y;
        if (z > 0.0) {
            z = 1.0;
        } else {
            z = z - x;
        }
        z = 2.0 * z;
        return z;
    }

    // The test driver
    public static void main(String[] args) {
        MyClassFP mc = new MyClassFP();
        double x = mc.myMethodFP(1.0, 22.0);
        Debug.printPC("\nMyClassFP.myMethodFP Path Condition: ");
```

```
    }
}
```

`MyClassFP.myMethodFP()` is the same as `MyClass1.myMethod()` except that everything is a `double`, not an `int`: Its code is in `extensions/symbc/examples/MyClassFP.java`.

The Eclipse run config parameters are:

```
+vm.insn_factory.class=gov.nasa.jpf.symbc.SymbolicInstructionFactory
+vm.classpath=.
+vm.storage.class=
+symbolic.method=myMethodFP(sym#sym)
+jpf.report.console.finished=
MyClassFP
```

And the resulting test cases are:

```
Test Case 1:  y = -4.99999991632194E-7,  x = -50.00000025000001
Test Case 2:  y = 0.0,                   x = 0.0
Test Case 3:  y = -4.99999991632194E-7,  x = 50.00000025000001
```

At this point you might wonder why there are three test cases instead of the two you got when the parameters were integers. The answer is that JPF operates on the JVM bytecodes, not the Java source code. When `z` is a floating point variable, the code

```
        if (z > 0.0) {
            z = 1.0;
        } else {
            z = z - x;
        }
```

is compiled to bytecodes that act as though the Java source was (conceptually, at least) something like

```
        if (z > 0.0) {
            z = 1.0;
        } else if (z == 0.0  ||  z < 0.0) {
            z = z - x;
        }
```

So you get one test case for `z > 0.0`, one for `z == 0.0`, and one for `z < 0.0`.

Object Fields

Instead of doing symbolic evaluation of method parameters, you might need to do it with fields of objects. One way to do this is to have your driver assign symbolic method parameters to the object's fields:

```
public class MyClassWithFields {
    public int field1;
    public int field2;

    public void myMethod1() {
    ...etc...
```

```
    }
}

public class MyDriverForFields {
    private static void makeFieldsSymbolic(int x, int y) {
        MyClassWithFields mc = new MyClassWithFields();

        mc.field1 = x;
        mc.field2 = y;
        mc.myMethod1();
        Debug.printPC("\nMyClassWithFields.myMethod1 Path Condition: ");
    }

    // The test driver
    public static void main(String[] args) {
        makeFieldsSymbolic(1,2);
    }
}
```

This, along with these Eclipse run config parameters:

```
+vm.insn_factory.class=gov.nasa.jpf.symbc.SymbolicInstructionFactory
+vm.classpath=.
+vm.storage.class=
+symbolic.method=makeFieldsSymbolic(sym#sym)
+jpf.report.console.finished=
MyDriverForFields
```

will do the trick.

The other way to do this is with the `@Symbolic()` annotation:

```
public class MyClassWithFields {
    @Symbolic("true")
    public int field1;

    @Symbolic("true")
    public int field2;

    public void myMethod1() {
        int z = field1 + field2;
        if (z > 0) {
            z = 1;
        } else {
            z = z - field1;
        }
        z = field1 * z;
        return z;
    }
}
```

```
public class MyDriverForFields {
    // The test driver
    public static void main(String[] args) {
        MyClassWithFields mc = new MyClassWithFields();

        mc.myMethod1();
```

```
        Debug.printPC("\nMyClassWithFields.myMethod1 Path Condition: ");
    }
}
```

and with these Eclipse run config parameters:

```
+vm.insn_factory.class=gov.nasa.jpf.symbc.SymbolicInstructionFactory
+vm.classpath=.
+vm.storage.class=
+symbolic.method=myMethod1()
+jpf.report.console.finished=
MyDriverForFields
```

will produce the same results as above, namely:

```
Test Case 1:  field2 = -9999999,  field2 = 10000000
Test Case 2:  field2 = -10000000, field2 = -10000000
```

As usual, these classes are in `extensions/symbc/examples/MyClassWithFields.java` and `extensions/symbc/examples/MyDriverForFields.java`.

A Larger Example: Code Coverage and Validation

Suppose you not only need test cases for each path through your code, but for each test case you also need to know which path it covers. This can be done easily enough by annotating your code. However, whenever you annotate your code (i.e. change it) you run the risk of introducing bugs. (We skipped over this problem in the earlier annotation example.) So we need a way of verifying that the test cases we generate have the same effect on the original code as they do on the annotated code.

To make all this happen we will do the following:

- Annotate each "path segment" in the code (typically, each clause of an `if` statement) with an identifier for that segment.
- Run the test case on the original code and verify that we get the same results

Our original code consists of a bunch of `if` statements. We will keep track of the path by assigning an ID to each branch of each `if` statement and appending that ID to a string variable when the segment is executed.

To verify that each test case produces the same result in both versions of the code we will extract the symbolic variable values from the path condition and run them through both versions, comparing the results.

Finally, we will print the path taken by each test case along with the input values.

This is the original code, in the package `coverage`:

```
package coverage;

public class MyClassOriginal {
    public int myMethod(int x, int y) {
        int z = x + y;
        if (z > 0) {
            z = 1;
```

```
        } else {
            z = z - x;
        }

        if (x < 0) {
            z = z * 2;
        } else if (x < 10) {
            z = z + 2;
        } else {
            z = -z;
        }

        if (y < 5) {
            z = z - 12;
        } else {
            z = z - 30;
        }
        return z;
    }
}
```

Now for the code with the annotations to identify the path taken. The variable `path` holds the string describing the path taken. For the "path segment" IDs, we use human-readable descriptions like `"z>0 "`. At the end of `myMethod()` we have also added a call that validates and prints the test case: `CheckCoverage.processTestCase()`. `CheckCoverage` is discussed in more detail later.

```
package coverage;

public class MyClassWithPathAnnotations {
    public int myMethod(int x, int y) {
        StringBuilder path = new StringBuilder();
        int z = x + y;
        if (z > 0) {
            path.append("z>0 ");
            z = 1;
        } else {
            path.append("z<=0 ");
            z = z - x;
        }

        if (x < 0) {
            path.append("x<0 ");
            z = z * 2;
        } else if (x < 10) {
            path.append("0<=x<10 ");
            z = z + 2;
        } else {
            path.append("x>=10 ");
            z = -z;
        }

        if (y < 5) {
            path.append("y<5 ");
            z = z - 12;
        } else {
            path.append("y>=5 ");
            z = z - 30;
        }
        CheckCoverage.processTestCase(path.toString());
```

```
        return z;
    }
}
```

And this is the driver:

```
package coverage;

public class MyDriverForPathAnnotations {
    // The test driver
    public static void main(String[] args) {
        MyClassWithPathAnnotations mca = new MyClassWithPathAnnotations();
        mca.myMethod(1, 2);
    }
}
```

The trickiest part here is `CheckCoverage.processTestCase()`. To understand this part, you will have to recall that when you run your programs under JPF, there are actually two VMs involved. Your code runs under JPF's VM, but JPF runs under a native JVM. You use JPF's "native interface", MJI, to comunicate between the two VMs.

You need to know this because we will need to get the current path condition, extract from it the current variables and their values, validate those values, and print the test cases. But the current path condition and its class (`gov.nasa.jpf.symbc.numeric.PathCondition`) do not exist in the JVM your code is running in. Your code runs in the JPF VM, but path conditions exist only in the base JVM, i.e. the Sun JVM in which JPF itself runs.

With that in mind, here is the code for `CheckCoverage`. As always with MJI, there are two parts: a declaration on the JPF VM side for class `CheckCoverage` and an implementation on the base JVM side for the peer class `JPF_coverage_CheckCoverage`.

```
package coverage;

public class CheckCoverage {
    public static void processTestCase(String path) {}
}
```

```
package coverage;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import gov.nasa.jpf.jvm.MJIEnv;
import gov.nasa.jpf.symbc.JPF_gov_nasa_jpf_symbc_Debug;
import gov.nasa.jpf.symbc.numeric.PathCondition;

public class JPF_coverage_CheckCoverage {
    private static Set paths = new HashSet();
    private static int caseNum = 0;

    public static void processTestCase(MJIEnv env, int objRef, int pathRef) {
        String path = env.getStringObject(pathRef);
```

```
        if (! paths.contains(path)) {
            paths.add(path);
            Map varsVals = getPcVarsVals(env);
            int x = (Integer)varsVals.get("x");
            int y = (Integer)varsVals.get("y");
            int origZ = new MyClassOriginal().myMethod(x, y);
            int annoZ = new MyClassWithPathAnnotations().myMethod(x, y);

            caseNum++;
            if (origZ == annoZ) {
                System.out.format("TestCase %s:  x = %s  y = %s\nPath: %s\n\n",
                        caseNum, x, y, path);
            } else {
                System.out.format("Error for TestCase %s:  x = %s  y = %s\nPath: %s\n\n",
                        caseNum, x, y, path);
                System.out.format("The annotated and original code got different
results.\n" +
                        "Annotated Result: %s\n" +
                        "Original Result: %s\n\n",
                        annoZ, origZ);
            }

        } else {
            System.out.println("Already saw '" + path + "'");
        }
    }

    private static Map getPcVarsVals(MJIEnv env) {
        Map varsVals = new HashMap();
        PathCondition pc = JPF_gov_nasa_jpf_symbc_Debug.getPC(env);

        if (pc != null) {
            pc.solve();
            pc.header.getVarVals(varsVals);
        }
        return varsVals;
    }
}
```

All of this code is in `extensions/symbc/examples/coverage`. Here is the Eclipse config data needed to run it:

```
+vm.insn_factory.class=gov.nasa.jpf.symbc.SymbolicInstructionFactory
+vm.classpath=.
+vm.storage.class=
+symbolic.method=myMethod(sym#sym)
+jpf.report.console.finished=
coverage.MyDriverForPathAnnotations
```

And here is what you get for output when you run it:

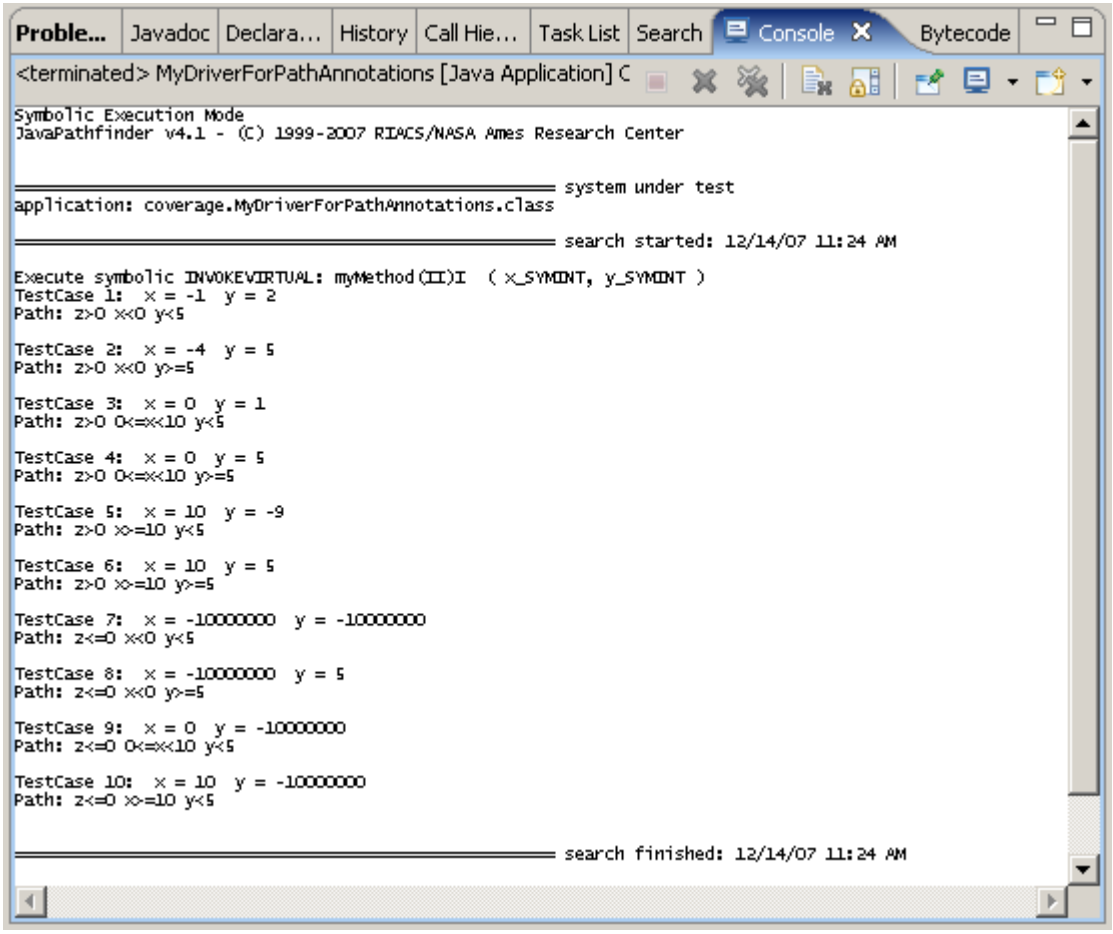

Figure 5: Eclipse Output With Path Annotations

One thing you might notice here is that there are only ten test cases. But if you do the math there should be `2*3*2=12`. The two missing test cases would have the paths `z<=0 0<=x<10 y>=5` and `z<=0 x>=10 y>=5`. It turns out these cannot happen: at the first `if` statement, `z = x + y`, so if you work out the inequalities you will find there are no values for `x` and `x` that will generate these paths. The cool thing is that JPF and Choco figure this out automatically.

Other Examples

The test directory, `extensions/symbc/test/gov/nasa/jpf/symbc`, has lots of examples but they cover pretty much the same ground as the ones above.

Common Problems

If you do not see a line at the start of your output similar to the one labeled *myMethod() is called symbolically* in Figure 4, then your method is not being called symbolically. That usually happens because you have the wrong value for `symbolic.method` in your program arguments. Make sure that you have both the right method name and the right number of arguments in your Eclipse run configuration.

If JPF seems to go into an infinite loop, it may be because the Choco constraint solver cannot find a solution for one of your path conditions.

Depending on when you last updated your copy of JPF from SourceForge, floating point symbolic execution may or may not work for you -- we are still working on it.