Introduction to Java PathFinder

Raghavan Komondoor

Indian Institute of Science, Bangalore

In this document we give an introduction to Java PathFinder (JPF), and its symbolic variant, which is called Symbolic JPF or SPF. We also give an overview of the working principles of JPF and SPF, along with an illustration.

1 Objective of Java Path Finder (JPF)

JPF is meant to automatically generate numerous runs of a given program. In other words, it is an automated, exhaustive testing tool. It aims to cover as many parts of the program as possible. It also aims to find runs that cause exceptions wherever exceptions are possible. If the programmer has added assertions, it tries to find runs that cause the assertions to fail. We discuss in Section 3 how JPF can be used for functional testing as well.

2 Working principle of JPF

JPF is based on a modified Java Virtual Machine (JVM). The main additional feature in this modified JVM are the notions of *choices* and *backtracking*. For instance, say we have a statement "`int v = random.nextInt(5)`" in the program. In normal execution this statement returns a random number between 0 and 4. In JPF, this statement is treated as a *choice point*, and a set of choices are generated when this statement is first visited. The choices correspond to all possible values (i.e., 0 to 4). One of the choices generated is immediately taken, and the other choices are saved on a stack. The system's state (run time stack and heap) is fully check pointed when a choice is taken. Whenever control reaches the end of the program (i.e., end of main()), control backtracks to the most recent choice point with unexplored choices and the system's state is restored to the state that was check-pointed when the choice point was first reached. Then, one of the remaining choices is taken. JPF terminates when no more unexplored choices remain in any choice point (see the example program Rand.java in jpf-core/src/examples).

In other words, JPF enables multiple runs of a program to be exhaustively explored within a single invocation.

3 Symbolic JPF

Symbolic JPF (SPF) is an extension of JPF that eliminates the need for the programmer to create choice points explicitly using calls to random.nextInt(). In SPF, certain conditionals in the program automatically get treated as choice points, with two choices: *true* and *false*. This is achieved using an idea called *symbolic execution*. We explain symbolic execution in SPF briefly in this section. For now, let us assume that we have only scalars in the program (i.e., no pointers, objects, or memory allocation).

Execution begins from main(), and initially stays in a concrete mode (i.e., with concrete values in all variables and heap locations). Whenever control enters any of the methods specified in the `symbolic.method` option in the .jpf file (these methods are called methods-under-test or MUTs), each formal parameter that is marked *sym* in the .jpf file will be assigned an *initial symbolic value*, while the parameters that are marked *con* will get the actual concrete values that are passed from the call site. The method body will now be executed. If any variable in the RHS of an assignment statement has a symbolic value or symbolic expression (i.e., is not concrete), the LHS variable will automatically be assigned a symbolic expression. Note, the only symbols that will ever be used in symbolic expressions are the initial symbolic values of the formal parameters (unless fresh symbols are created programmatically, which we discuss later).

A definition: A *state* of the program is a mapping from the set of all variables to symbolic expressions or concrete values. That is, it is a mapping that maps each variable to a symbolic expression or concrete value.

At each step during the execution, the modified JVM of SPF maintains the current *path condition* (PC). This is a boolean formula in terms of the symbols introduced so far. When the execution enters a method-under-test (MUT) for the first time, the PC is set to *true*. Whenever a conditional is reached, there are two scenarios. The first scenario is that as per the current state of the program all variables that occur in the conditional have concrete values. In this scenario, the conditional is not treated as a choice point, execution proceeds down the appropriate branch.

The other scenario is that some of the variables in the conditional have symbolic values or expressions as per the current program state. In this scenario the conditional is treated as a choice point. The syntactic condition in the program is converted into a symbolic condition using the current state of the program, and this symbolic condition is the first choice. The other choice is the negation of this symbolic condition. One of these choices is saved for later, and the other is immediately considered. This considered choice is first conjuncted with the current PC to yield a new strengthened PC. If this new PC is satisfiable, then the considered choice is taken and execution proceeds along the branch that this choice corresponds to with the strengthened PC. On the other hand, if the strengthened PC is unsatisfiable, then this choice is dropped and the other choice is considered.

Whenever backtracking happens to a choice point, the state of the program as well as the PC are reverted to what they were the first time this point was reached, and then the other choice (which was previous saved) is considered.

Note, if the return variable of a MUT has a symbolic value (as opposed to a concrete value), then in the calling method the variable that gets assigned the return value will also have a symbolic value now rather than a concrete value. If this variable is used to compute the values of other variables, they will become symbolic as well. If any of these variables is used in a conditional (the conditional could be outside any of the MUTs), this conditional will become a choice point. In other words, symbolic execution and choice points are not necessarily restricted to occur within the body of the MUTs.

Note that whenever a variable or a field of a concrete object holds a symbolic value or symbolic expression, JPF also maintains a concrete value for this same variable or field. However, these concrete values are basically garbage. Other than print statements, no other statements can access these values. If a symbolic expression resides in an integer variable `v`, this expression can be obtained in the program as a string using the call `Debug.getSymbolicIntegerValue(v)` (analogous calls exist for other types). The call `Debug.getSolvedPC()` returns the PC at the current point along with a solution to the symbols that appear in the PC.

Here is a suggestion on how to use SPF for functional testing. Consider a sorting function that sorts a given array in-place. Our objective is to check if there are any bugs in this function. The argument array to the sorting function can be marked as "concrete" in the .jpf file. In the calling function, the programmer can create of an array (of some fixed size), and place a symbol (not a concrete value) in each element of the array. For instance, an integer symbol can be created using the method `Debug.makeSymbolicInteger()`. This array is to be passed to the sorting function. The programmer then writes a loop after the call to the sorting function to check that the updated array is in sorted order. This loop could invoke an `assert(false)` statement if a sortedness violation is detected in the returned array. SPF will now try to find a run of the program that causes this assert statement to execute, and such a run, if found, would indicate a bug in the sorting function.

4 Symbolic execution with pointers and objects

SPF allows formal parameters that are pointers to be also marked *sym*. For this to work correctly, one needs to assign the option *symbolic.lazy* the value *true* in the .jpf file. When a MUT is entered, corresponding to each formal parameter f_i that is marked *sym*, an initial symbol m_i is introduced and assigned to f_i in the program's state. Unlike what was mentioned in Section 3, additional symbols may get generated during the remainder of the execution. The names of these symbols would be of the form $m_i.f.\ldots.k$. Intuitively, such a symbol (which we call a *symbolic access path*) refers to the initial symbolic value residing in the memory location referred to by $f_i.f.\ldots.k$ when execution enters the MUT. Note, since f_i is marked *sym*, the actual concrete object it refers to and objects reachable from this object are ignored during the symbolic execution. Instead, f_i is treated as if it points to an object all of whose fields store symbolic values, the pointer fields of this object are further assumed to point to objects all of whose fields store symbolic values, and so on. These (virtual) objects are not all created when the MUT is entered. Rather, longer and longer symbolic access paths of the form $m_i.f.\ldots.k$ get created on-demand (i.e., lazily) as symbolic execution proceeds, as discussed in detail below.

In the presence of heap, the program *state* is extended to record the current contents of (i.e., values or symbolic expressions residing in) symbolic access paths in addition to the current contents of variables.

Numeric conditionals generate choices as described in Section 3. On the other hand, conditionals that compare pointers to other pointers or to null do not directly generate choices. Instead, each occurrence of a pointer in a conditional or in the RHS of a copying statement of the form "`w = v`" *may* create a choice point, as discussed in the rest of this paragraph. Say pointer variable `v` occurs in the RHS of a copying statement or in a conditional, and say `v` currently stores a symbol (or symbolic access path) ap_i as per the program state. If ap_i is constrained to be equal to some constant value as per the current PC, then execution proceeds normally and no choice point is generated. Otherwise, this occurrence of `v` is treated as a choice point, and generates the following choices: (i) ap_i is equal to null, (ii) ap_i equal to ap_j, for each other pointer-typed symbolic access path ap_j that has already been introduced so far, and (iii) ap_i is not equal to any other pointer-typed symbolic access path that has been introduced so far (this choice is expressed by SPF by selecting a fresh integer constant and equating ap_i to this constant in the PC). These choices are then considered one by one (with the help of backtracking, of course).

Get-field statements are processed in the following way. Consider the statement "`t = v.f`". If `v` refers to a concrete object or if `v` stores a symbol ap_i and $ap_i.f$ is assigned a concrete value in the program's state, then this get-field statement is executed normally and nothing more is to be done. (Note, `t` could still be assigned a symbolic value due to this statement if `v` refers to a concrete object and this object stores a symbolic value or expression in its 'f' field.) On the other hand, if the scenario mentioned above does not hold, then the following steps are taken:

1. Let ap_i be the symbol residing in `v` as per the program state.
2. If $ap_i.f$ is mapped to a symbolic value or expression p in the program state then

   Assign p to the variable `t` in the program state

   else if any synbolic access path ap'_i exists such that ap'_i is equal to ap_i as per the PC and such that $ap'_i.f$ is mapped to a symbolic value or expression q in the program state then

   Assign q to the variable `t` in the program state

   else

   Assign the symbolic access path $ap_i.f$ to the variable `t` in the program state. (*Note*: this is the only situation in which a new, longer symbolic access path gets introduced during symbolic execution.)

3. Let r denote the symbolic expression that was assigned to `t` in the previous step (i.e., r is equal to p or q or $ap_i.f$). If r is already participating in an equality constraint as per the current PC, then there is nothing more to do. Otherwise, a choice point is generated corresponding to the occurrence of variable `v` in the current statement "`t = v.f`". The choices are (i) r is equal to null, (ii) r is equal to each other symbolic access path that been introduced so far (one at a time), and (iii) r is not equal to any other pointer-typed symbolic access path that has been introduced so far (i.e. r is equal to a fresh integer constant). These choices are then considered one by one (with the help of backtracking, of course).

When a put-field statement "`v.f = t`" is executed, if `v` refers to a concrete object then value in `t` is just copied to the '`f`' field of this object (irrespective of whether `t` stores a concrete or a symbolic value). Otherwise, if `v` stores a symbol ap_j and if `t` has a symbolic expression or value n, $ap_j.f$ is assigned n in the updated program state.

Allocation sites (i.e., calls to "new") are handled concretely. That is, they return normal concrete objects. However, there is a way for the programmer to change the way an allocation is invoked in the program to make it create a symbol instead of a concrete object.

Note that in general a run may not terminate and longer and longer symbolic names may be generated. This would happen, e.g., if a loop traverses a symbolic list (i.e., starts traversal from a symbolic head pointer). In such cases termination would be assured only if concrete-valued variables are used to bound the number of iterations of loops.

5 Interpreting the output from the tool

We now illustrate the approach followed by SPF using the example below. The method main() (not shown below) invokes the method test3, passing two concrete objects.

```
    public static void test3(NodeSimple n, NodeSimple m) {
42:   if (n == null || m == null) return;
43:   n.next = m;
44:   if (n.next.next == null)
45:     System.out.println("Problem!");
46: }
```

We run the tool with both arguments of the function above marked as *sym*, using the listeners *gov.nasa.jpf.symbc.SymbolicListener* and *gov.nasa.jpf.listener.ExecTracker*, with the *symbolic.lazy* and *symbolic.debug* options set to true. The options *et.print_src* and *et.print_mth* are set to *true*, while the option *et.print_insn* is set to *false*.

A tree representation of the tool's output for the example above, which is generally called a *symbolic execution tree*, is depicted in Figure 1.

We now focus on the tool's actual output. At regular intervals in the output, the current line number location in the program is printed. When control enters the method `test3`, we see the following chunk of output:

```
New sym int n_1_SYMREF min=-2147483648, max=2147483647
New sym int m_2_SYMREF min=-2147483648, max=2147483647
```

This corresponds to creating the symbols n_1 and m_2 and assigning them to the formal parameters `n` and `m`. This is also depicted at the top of Figure 1.

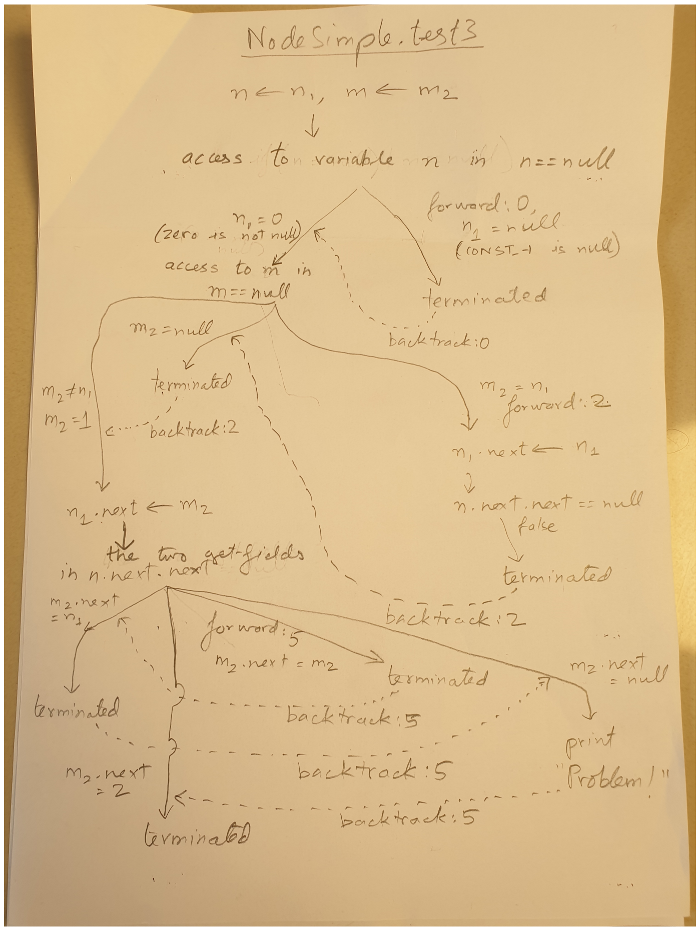

Figure 1: Symbolic execution tree

Each time a choice point is reached *for the first time*, a message of the form "`forward:` *num* new" is printed, where *num* is a number that identifies the choice point. In the example, the access to `n` in the conditional "`n == null`" creates a choice point, and this is marked in the output via a message "`forward: 0 new`". Thus, this is the choice point 0. The first choice at this point is $n_1 = null$. The PC after this choice is made is printed as "`n_1_SYMREF = CONST_-1`" just below `forward: 0`. Note, the value -1 represents null in JPF. This choice corresponds to the rightward-going branch just below the root of the tree in Figure 1.

This choice results in control immediately returning to main(), and then to the end of the program (since main() has no code after the call to test3). This is indicated by the "`thread terminated`" message. Right after this message there is a "`forward: 1 new end`" message, which indicates that termination is reached. We then see the message "`backtrack: 0`", which means control is backtracking to the choice point 0 and the next choice is being considered. We then see the following message:

```
n_1_SYMREF = CONST_0 &&
n_1_SYMREF != CONST_-1
```

This means the PC after the second choice is taken at choice point 0 is $n_1 = 0$. Here, zero is a fresh constant value chosen by SPF, and this constraint indicates that n_1 is not equal to any other symbol. Note, SPF uses the constant value -1 and not zero to indicate null. Now, since in this choice n_1 is not null, we also see the following messages, which indicate the introduction of the symbols $n_1.elem$ and $n_2.next$:

```
New sym int n_1_SYMREF.elem min=-2147483648, max=2147483647
New sym int n_1_SYMREF.next min=-2147483648, max=2147483647
```

The backtracking mentioned above and the second choice that was then taken at choice point 0 is depicted as the left branch from the root node in the tree in Figure 1.

Now control reaches the reference to variable `m` in the conditional "`m == null`". This generates a choice point (with ID = 2), and a "`forward:  2 new`" message. After this forward message one sees the following message:

```
m_2_SYMREF = n_1_SYMREF &&
n_1_SYMREF = CONST_0 &&
n_1_SYMREF != CONST_-1
```

This is the updated PC after the first choice is taken at choice point 2. From this we can infer that the choice taken is $m_2 = n_1$. The run now continues to the put-field statement in Line 43 of the code. Here, the program state is updated to make $n_1.next$ have the value m_2, which itself is equal to n_1 as per the PC. Therefore, when the two get-fields in Line 44 are reached, both `n.next` and `n.next.next` are both determined to be equal to n_1 itself as per the program state in conjunction with $m_2 = n_1$ in the PC. Since n_1 is already participating in an equality constraint as per the current PC (namely, $n_1 = 0$), no further choices need to explored to see what these two field accesses are equal to, and hence no choice point is generated here. Control does not enter Line 45, since n_1 is not null as per the PC. Control then leaves test3(), and then main(). The "`thread terminated`" message appears, and then there is a "`backtrack:  2`" message.

Everything mentioned above appears in the rightmost subtree of the node labeled "access to `m` in the conditional `m == null`" in Figure 1.

The middle subtree of the node mentioned above corresponds to the second choice, i.e., $m_2 = null$, at choice point 2.

The left subtree of the node mentioned above corresponds to the choice that m_2 is not equal to any other symbol so far. Therefore, m_2 is taken as equal to a fresh constant 1 (recall that n_1 has been equated to the constant zero). In the left sub-tree, `n.next` is assigned m_2 due to Line 43 of code. Two get-field operations are now encountered. `n.next` is equal to m_2 due to Line 43, and there already exists an equality constraint $m_2 = 1$ in the PC. However, `n.next.next`, which is equivalent to $m_2.next$, is unconstrained both in the program state and in the PC so far. Therefore, this get-field becomes a choice point, with ID = 5. The choices are basically to equate $m_2.next$ with null, with every other symbol introduced in the path so far (i.e., n_1 and m_2), and finally with a fresh constant (namely, 2). Thus, there are four choices. See the node labeled "the two get-fields in `n.next.next`" in Figure 1, which has four child subtrees corresponding to the four cases mentioned above.

The first choice corresponds to the third sub-tree from left of the node mentioned above. (The order of subtrees of any node in the figure is arbitrary.) The PC after this choice is taken occurs right after "`forward:  5`" in the output. The PCs after the second, third, and fourth choices are taken, respectively, are printed right after the first, second, and third occurrences of the messages "`backward: 5`" in the output, respectively. After this no more unexplored choices remain in any choice point, and JPF terminates.

The example considered in this section did not contain any numeric conditionals. If there were any, each numeric conditional would have been treated as a choice point. There are a couple of differences in how the choices for numeric choice prints are printed in the output file compared to choice points due to pointers: (a) The PC after the first choice in a choice point is taken gets printed *before* and not after the corresponding `forward:` message. (b) Before any `forward:` message, all the choices that are to be considered for that choice point get printed. Each choice is prefixed by a "`### PCs`" string. (c) After each backtrack, the PC after the next choice is taken is *not* printed. However, the "`backtrack:` n" messages still appear.