Symbolic backward simulation of Java bytecode program
Tetsuya Inafune*, Shinichi Miura, Toshihiro Taketa, Yukio Hiranaka
Yamagata University
4-3-16 Jonan, Yonezawa
Yamagata, Japan
+81-238-26-3322
zioi@acm.org (corresponding author)  *currently NTT DATA i Corporation

ABSTRACT
We present a new method, symbolic backward simulation, for detecting bugs in Java bytecode programs. In order to find bugs comprehensively, the method determines conditions on the input side by tracing back from the tail of the program while performing reverse execution for each bytecode. Generally, reverse execution is difficult, especially for instructions of two-input-one-output operations and branches. Our method solves the problem symbolically with essentially fewer simulation cases than numerical testing and forward symbolic analysis. We also show simulation results which detected a branch condition error and a real number processing error.

CCS Concepts
• Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging.

Keywords
symbolic analysis; backward simulation; Java bytecode; reverse execution; bug detection

1. INTRODUCTION
Unintentional operations of software systems, such as those caused by bugs, may have a severe influence on our society. Typical software testing (Figure 1) cannot sufficiently eradicate them because it can only guarantee the behavior under the conditions tested.

However, output values are often limited to a small number of choices. As an example, Figure 2 shows a program that compares two input values and outputs 1 or 0 depending on the relation between the two integer inputs. The number of input combinations is \( N^2 \) (\( N \) is the total number of integer values), which is larger than the number of possible integer outputs (\( N \)). Actually, the number of output cases may be very small as in the Figure 2. In such conditions, if we examine the program starting from outputs and tracing backward, we can reduce search cases [1-3]. Moreover, we can limit our traces only to cases of output values concerned.

Backward tracing needs reverse execution. Various kinds of reverse execution have been considered mainly for debugging [4-8]. One is the checkpoint/rollback method for enabling reverse execution from the point where a problem occurred by using state logs [8]. In another method, equivalent reverse codes are generated to reduce the size of logs [4]. Both methods are for debugging only and exhaustive detection cannot be performed.

We have made a numerical backward range simulator to obtain a possible input range for a given output value [3]. It uses a method to divide input/output values into finite ranges to perform the simulation efficiently. The resolution of the result can be arbitrarily improved by narrowing the ranges of values. Also, it can take into account the problem of overflow of arithmetic instructions.

![Figure 1. Typical forward software testing.](image1)

![Figure 2. Many input cases and small number of output cases.](image2)

![Figure 3. Numerical backward range simulation [3].](image3)

Figure 3 is an illustrative schematic of numerical backward simulation where integer inputs are divided into four ranges. Two
dimensional relations between input ranges corresponds to outputs 1 and 0, and roughly showing relations $x \geq y$ and $x \cdot y$, respectively. By reducing the range width, the graphs converge to those equations. If the simulation starts with an output value other than 0 or 1, the simulation will stop because there is no possibility of such an output value through the program. The arrows with broken lines indicate backward tracing, and the arrows with solid lines indicate forward tracing (normal program execution) in the figures in this paper.

In the numerical range simulation, improving the resolution will increase the number of cases to be traced and the calculation time will increase exponentially. On the other hand, symbolic analysis examines the conditions of all program branches and determines symbolic equations for each terminal condition (Figure 4) [6, 9-11]. The number of terminal conditions does not exceed the number of program branches plus 1. Although, the number of terminal conditions for a program loop may be unknown beforehand and might be infinite.

Symbolic execution has a promising feature of limiting cases to be considered. Therefore, we intend to use it in a backward simulation. Figure 5 illustrates backward symbolic execution, in which the simulator determines the input relations expressed in the equations by tracing backward from a certain output value. We call the tracing as symbolic backward simulation because there may be multiple output cases and multiple branch cases which must be traced individually. Since we have to test only the cases that match the starting output, we do not need to evaluate all cases unlike the forward symbolic analysis.

The composition of this paper is in the order of basics of symbolic backward simulation, implementation of the backward simulator, verification of the simulation with two examples, discussions and conclusion.

![Figure 4. Forward symbolic analysis.](image)

**Figure 4. Forward symbolic analysis.**

| symbolic analysis | if($x \geq y$) $z=1$
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>else $z=0$</td>
</tr>
<tr>
<td></td>
<td>$x \cdot y$, $z=1$</td>
</tr>
<tr>
<td></td>
<td>$x &lt; y$, $z=0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input $X,Y$</th>
<th>output $Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \geq y$</td>
<td></td>
</tr>
<tr>
<td>$x &lt; y$</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 5. Backward symbolic execution.](image)

**Figure 5. Backward symbolic execution.**

2. SYMBOLIC BACKWARD SIMULATION

2.1 Reverse Execution and Symbolic Values

Most Java bytecodes handle only the operand stack and local variables. In forward execution, they transform the pre-execution state into the post-execution state (Figure 6). Then, reverse execution can be done by transforming the post-execution state back to the pre-execution state. However, undefined variables and unknown entry on the operand stack may appear through the backward simulation. So we express their values symbolically as shown in Table 1.

![Figure 6. Forward and reverse bytecode execution.](image)

**Figure 6. Forward and reverse bytecode execution.**

<table>
<thead>
<tr>
<th>Table 1. Symbolic expression of variables.</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable type</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>$L_m(n)$</td>
</tr>
<tr>
<td>$A_{R_m}$</td>
</tr>
<tr>
<td>$A_m$</td>
</tr>
<tr>
<td>$V_n$</td>
</tr>
</tbody>
</table>

We explain how our reverse execution works with an example in Figure 7. It shows a small segment of Java bytecodes, in which iconst_1 loads an integer constant value 1 onto the operand stack, and istore_1 saves it in the local integer variable 1 [12]. Figure 7(b) shows the corresponding backward execution of those two instructions. In the backward execution, it is necessary to distinguish between the pre-execution state and the post-execution state. Two lines corresponding to each instruction show the post-execution state on the first line and the pre-execution state on the second line (starting with //). In the backward execution of istore_1, the value of local variable 1 is returned to the operand stack. However, the value of local variable 1 is unknown in the post-execution state, then it is represented symbolically as $L_1(0)$. The number in parentheses increases by 1 each time its value has a possibility of change from the last expressed value.

![Figure 7. Examples of forward and backward bytecode execution.](image)

**Figure 7. Examples of forward and backward bytecode execution.**

2.2 Symbolic Determination

By continuing reverse executions, unknown values may be determined or restricting conditions may become known. In Figure 7(b), $L_1(0)$ on the operand stack in the pre-execution state of istore_1 must coincide with the value 1 on the operand stack in the post-execution state of iconst_1. Therefore, $L_1(0)$ must be 1. If $L_1(0)$ is not 1, it shows that the simulated case is not feasible.

2.3 Branch Processing

Branch processing is necessary in the backward simulation if there is a branch instruction in the program. In Figure 8(a),
branching to the 11th bytecode may occur after executing the 4th bytecode in the forward branch execution (ifle 11). In the backward simulation in Figure 8(b), a backward branch point (indicated by circled j) is provided before the 11th bytecode, and both possibilities of returning to the 10th bytecode and to the 4th bytecode are indicated by the control flows. As we cannot know which branch occurred at this point, we must trace both of the branches. By tracing further, we will know which one is valid or both cases are valid. Such a branch search is performed in depth-first manner by our simulator.

Figure 9 shows the details of those branch cases. Assuming that the branching to the 11th bytecode did not occur and proceeded to the 7th line (Figure 9(a)), the value V1 for ifle instruction must be positive (therefore L1(1)> 0). Otherwise, when branching to the 11th bytecode from the 4th bytecode occurred (Figure 9(b)), V1 must be 0 or less (therefore L1(0)<=0). Tracing backward carrying known conditions, it will be found that the value is 10 at the 0th bytecode. The case without branching is successful and the case with branching fails. If any contradictory condition appears at the end or along the way, the case traced must be rejected. And the case reached the start of the program without any contradiction remains as a final condition.

2.4 Loop Processing
A loop may occur even in a backward execution as well as in a forward execution. In such a case, the number of passes within the loop is unknown in the backward simulation. Try and determine is the only method for determining whether branching at the beginning of the loop, exiting the loop, or iterating the loop occurred. As an example in Figure 10(b), there is a matching case after exiting the loop (the case when x=0). In general, there is no way to determine how many loop passes must be tried, so we should set a sufficiently large number as the maximum loop passes to terminate the simulation within a finite time.

![Figure 8. Forward and backward execution of a Java program with a branch bytecode.](image)

**3. IMPLEMENTATION**

Figure 11 shows the processing outline of the symbolic backward simulation. First, a bytecode instruction list is created from the target Java class file. Next, branch instructions are examined to get branch points for backward processing. Then, each instruction is executed and traced backward from the last instruction. When it reaches the start of the program, we will obtain the resultant input condition for the path taken. If there are remaining branch paths, the simulator traces all of them.

We implemented basic instructions (const, load, store, arithmetic operation, branch, goto, return, invoke instructions, around the half of instructions [12]) which we considered necessary for our current study. We also implemented a forward simulation mode for validation of the results obtained by the backward simulation.

![Figure 9. Backward flow of the branch section in Fig. 8.](image)

**Figure 9.**

![Figure 10. Forward and backward execution of a pseudo code loop.](image)

**Figure 10.**

![Figure 11. Control steps of a backward simulation.](image)

**Figure 11.**
Figure 12 is an example of state changes through the backward execution of a program which calculates \(1 + 10\). In the backward execution, five equations shown in the rightmost column are obtained. By processing them algebraically, we can get the final result that \(L_3(0)\) is equal to 11, correctly. This algebraic processing is executed inside the backward simulator.

### 4. VERIFICATION EXAMPLES

Here, we show two examples of bug detection. The first one, Figure 13, calculates \(y / x\) when the first input value \(x\) is \(0 < x <= 100\) and the second input value \(y\) is \(0 <= y <= 100\). This program works correctly for most combinations of inputs, but it outputs unexpected value “0” for the case of coincidence of the two inputs owing to the mistake “\(x > y\)" in the 9th line, which must be “\(x >= y\)" for correct output of “100”.

The result obtained by the backward simulation (Figure 14) with starting value of 0 as the output is shown in Figure 15. It indicates in the last line that \(L_1(0) \leq L_2(0)\) is the possible condition, which includes unintentional condition of \(L_1(0) = L_2(0)\) (which means \(x=y\)). This is an example in which input conditions covering multiple input patterns can be derived from one output value by performing our simulation.

The second program in Figure 16 is an example of numerical errors, which calculates how many pieces of a pie can be cut by subtracting the reciprocal of the integer divisor input \(x\). A loop exists between the 6th line and the 8th line. The loop is repeated as long as the variable \(z\) corresponding to the remaining pie is larger than 0. The value \(z\) is decreased by 1 / \(x\) every time it passes through the loop. Finally, the program outputs the value of \(y\) (expected to be \(x\)).

```java
public class Test {
public static void main(String[] args){
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    int z = 0;
    if(x > 0 && x <=100 ) {
        if(y >= 0 && y <=100) {
            if(x > y) {
                y = y * 100;
                z = y / x;
            }
        }
    }
    System.out.println(z);
}
```

Figure 13. Java source code of a percentile calculator.

Figure 14. Control flow of the backward simulation of Fig. 13.
If the input value is 3, the output will be 4 instead of 3. This is because the floating point value of \((\text{double}) 1 / x\) in the 6th line of Figure 16 becomes smaller than 1/3. Such a numerical bug is a common mistake.

We derived the input condition by conducting a backward simulation, starting with the output value of 4. As the program has a loop, its output may be different depending on the number of loop passes. Then, we designed the simulator to increase the loop count from 1 to the predetermined maximum count. As the result shown in Figure 17, 3 <= L1(0) <= 4 is the last result for the case. It means that the output becomes 4 when the divisor input is 3 or 4.

```
public class Test {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = 0;
        if(x>0) {
            for(double z=1;z>0;z=z-1/(double)x) {
                y = y+1;
            }
        }
        System.out.println(y);
    }
}
```

Figure 15. Final result of backward simulation of Fig. 14.

Figure 16. Java source code of pie divider.

```
L3@0 > 0.0
L3@0 > 0.0
V35@0 > V36@0 > 0.0
L3@0 @ V37@0 / V38@0 > 0.0
L3@0 @ 1.0 / (double)V39@0 > 0.0
L3@0 @ 1.0 / (double)L1@0 > 0.0
1.0 @ 1.0 / (double)L1@0
3 <= L1@0 <= 4
L3@0 > 0.0
L3@0 > 0.0
1.0 > 0.0
L1@0 > 0.0
final input condition
invokestatic #2
3 <= L1@0 <= 4
```

Figure 17. Final result of backward simulation of Fig. 16 by setting y=4.

5. DISCUSSION

From the results of the simulations, it was confirmed that backward symbolic analysis can clarify input conditions including multiple input combinations for Java bytecode programs. As the basis of the simulation, backward execution on arithmetic operations, conditional branches, and loops have been performed by using symbolic value substitution through the simulation.

Some problems remain to be solved in the symbolic backward simulation. Firstly, it is difficult to process overflow, though possible by using a numerical method [3], because symbolic methods assume correctness of bytecode instructions. Secondly, we should use the simulator as an imperfect but practical tool if the target has a loop which may cause an infinite loop. Currently, our simulator records loop count and stops after exceeding the predetermined count.

6. CONCLUSION

In this study, a symbolic backward simulator was implemented for Java bytecode programs and clarified to derive corresponding input conditions for specified output values. It was possible to derive input conditions covering multiple input patterns starting from an output value. Symbolic backward analysis is an effective method for bug detection. We expect to make it more useful by combining it with a numerical backward simulator properly.

7. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number 15K11989.

8. REFERENCES