ABSTRACT
In this paper we introduce a backward simulation of machine code program and report results of reduction methods of its processing time. It may be efficient to analyze a program by tracing back from the final result for detecting all the bugs caused by some codes in the program. We describe the backward simulator of Java bytecode program and methods of processing time reduction such as range division, stepwise resolution improvement, setting of search path, and exponential range division. Results of two short programs and effects of reduction methods are shown.

Keywords
backward simulation; Java bytecode; reverse execution; bug detection; exponential range division

1. INTRODUCTION
Failures of software system may make enormous effects on our society. To suppress such failures we need to eradicate bugs [9]. However, that is not easy because the most of testing tools are to detect bugs only on the conditions tested. To detect all the bugs and unexpected input conditions, it may be a good way to check possibilities of abnormal outputs backwardly. Number of test conditions will be greatly reduced by it because there is no need to test normal cases. Although, we need elaboration to perform reverse execution of software under test.

Reverse execution is studied and exists as practical tools [2][3] mainly for debugging such as a logging method in the forward execution for providing data to reverse execution after detecting a bug symptom, and a generating method of reverse program from forward program[1]. Unfortunately, the both methods enable testing only for some conditions and are not suitable to comprehensive detection of bugs.

We are developing a backward analyzer to determine the feasible input range for the specified output by using symbolic processing such as in [4][5][12] to be applied to numerical operations [10]. However, there must be difficulties for backward symbolic processing when complexity of the program under test increases. Then alternatively, we intend to try numerical backward simulation method which is already shown to be effective for the cases of system simulations [6][7].

In the following sections, we focus our simulation target to Java bytecode programs, which use a set of simple instructions. We describe outline of our backward simulation in section 2, how to execute each instruction reversely in section 3, reduction methods of processing time in section 4, implementation matters and branch control indispensable for the simulation in section 5, illustrative example of two simple backward simulation in section 6, and discussions and conclusion in the order.

2. BACKWARD RANGE SIMULATION
We show here outline of the backward simulation. The upper half of Figure 1 shows a simplified program which takes some input value to make corresponding output value with referring to its internal data or states. If we want to know the input value from the output value, it is natural to make a reverse program and to calculate by applying the output value to a backward input of the reverse program.

Of course, it may not be possible to determine input values from output values in many cases. Even if it is possible, the simulator may produce multiple possibilities of input values. Furthermore, application of a single value to the backward simulator cannot detect all the bugs. Then, range of output value (detailed description is in section 2) is introduced to the simulator as shown in the lower part of Figure 1. The resulting input value would also be a range of values, which can represent inclusion of multiple values.

Figure 2 shows a short and simple example of Java bytecode program which adds two immediate integer values (11 and 1). The left half is a forward process and the right half is a backward process. In the backward process, arrows show that directions of control flow are just reversed against the forward process, and broken lines explicitly show that they are backward control flows.

Figure 1. Forward simulation and backward simulation.
We implemented bytecode evaluator used it as a guessing engine comes that feasible that reverse add operation to get two simulation, which enables the simulation model in input single output instructions (inc, neg, type conversion, etc.), two input arithmetic instructions (add, sub, mul, div, rem), two input logical instruction (and, or, xor, shl, shr), and control flow instructions (if, goto, jsr, return, invoke, etc.).

Most of data operation instructions and single input single output instructions can be easily reversely executable. As a little bit detail of “bipush”, “iload_0” type and “istore_0” type, forward operation of “bipush” pushes the byte data of the instruction onto the operand stack, and reverse operation of “bipush” pops a range of values from the operand stack and check whether the range includes the instruction’s byte data. If the range includes the value, simulation control go upstream the program, otherwise simulation control stops with announcing that the simulated case is not feasible. Instructions of “iload_0” type are similar to “bipush” with modification of immediate value to local variable.

As forward operation of “istore_0” type instruction pops a value from the operand stack and put it in the 0th local variable, reverse operation of “istore_0” gets a range of values from the 0th local variable and put it onto the operand stack. If the value in the 0th local variable is not used before in the backward control flow, the stored value should be assumed with multiple cases of value range. Further in the backward simulation, feasibility of each assumed range will be determined.

In the cases of two input instructions which cannot be uniquely reversed, two input values can be expressed by divided multiple cases of range pairs, each of which is sequentially simulated further in depth priority search mode.

As for the backward operation of “iadd”, we have to deal with integer overflow, which may be a weak point for symbolic analysis tools and form a practical reason for backward simulation. Figure 3 shows input-output relation for the whole area of inputs.

3. REVERSE EXECUTION OF BYTECODE

Designing reverse execution needs consideration in detail for each bytecode. As the objective of our backward simulation is to detect bugs without exception, reverse calculation should stand on safe side which does not miss possibility of bug. We should utilize all the restricting conditions in the reverse execution to narrow down the resulting ranges because the range would naturally expand in the course of simulation process.

Here, we closely examine how to implement reverse execution of each bytecode. Java has 205 types of bytecode [11], which classified into data operation instructions (load, store, push, ldc, pop, dup, etc.), constant value input instructions (bipush, sipush, iconst_0, etc.), single input single output instructions (inc, neg, type conversion, etc.), two input arithmetic instructions (add, sub, mul, div, rem), two input logical instruction (and, or, shl, shr, ushr), and control flow instructions (if, goto, jsr, return, invoke, etc.).

Figure 2. Example of simple addition, (a) forward (b) backward.

Each byte code is treated as a simulation node and performs a simulation by receiving and sending backward information conveyed by JVM’s operand stack. We implemented bytecode nodes by bidirectional objects which can handle not only backward simulation but also forward simulation, which enables us to verify the results of backward simulation.

In a natural sense, we cannot reverse add operation to get two inputs from a single output value. However, we can perform it by doing simulations of possible cases of two inputs. As an example, in a case we can test 12 as a valid output or not for the program in Figure 2. If we suppose 1 for one input, the other input must be 11. Such relation holds for all input value pairs and the number of cases are finite for integer addition. The backward simulation in Figure 2(b) verifies the validity of the supposition. The bytecode “bipush 11” checks the backward information from “iadd” and passes it upward if it is 1, otherwise rejects the simulation flow. Subsequently, the upper bytecode “bipush 11” checks the other input value and passes it upward if it is 11, otherwise rejects the simulation flow. Therefore, only the backward information of 12 to “iadd” can pass through the simulation to the backward end point “start”. And we know by this simulations that feasible output of Figure 2 is 12 only. This is a theoretical way of backward simulation.

Practically, this method needs enormous time for performing huge number of repetition. Then comes the first idea to reduce processing time by testing a range of values instead of each value. As an example, starting by testing four pairs of nonnegative range and negative range upward from “iadd”, the simulation model in Figure 2(b) will instantly reject a range of negative value at “bipush 11” and “bipush 11”. Then, only a pair of two ranges of nonnegative values remains to pass. The area of possible value range is squeezed to a fourth.

Similarly, possible full range of integer value (from -2^{31} to 2^{31}-1) supposed for the backward input to “iadd” can be divided by two, four, eight and so on, continuing to a single value of 12. The width of such ranges as 2^{31}, 2^{30}, 2^{29} . . . represents the resolution of backward simulation. Lower resolution results can be obtained in a short time, higher resolution results requires longer processing time. Available processing time is the parameter to determine the resolution. However, usually, possibility under some resolution may be enough for results of backward simulation because human evaluator used it as a guessing parameter.

We introduce a parameter “nDIV” as 2 to the power of integer to be used throughout the simulator for determining the range by dividing the whole range of each value type.

Figure 3. Result of iadd, including overflow region.
In the upper right area and the lower left area outside the broken lines show overflowed area causing wrapping around effect.

To illustrate the mechanism of range operation used in our backward simulation, we explain a simplified model of integer addition for a case of 3 bit unsigned integer values (0-7). If we divide the whole integer range into four, we are to process any value by one of four ranges of [0, 1], [2, 3], [4, 5], [6, 7]. A forward addition of 0+7 is expressed by [0, 1] + [6, 7] and the result is the range of [6, 8]. As the value 8 does not exist in this integer definition, the range may treated as [6, 7] when omitting overflow, otherwise it should be treated as the possible ranges of [6, 7] and [0, 1].

As for the reverse “iadd” operation of [6, 7], there are following possible cases of two input values,

- [0, 1] + [6, 7]  
- [2, 3] + [4, 5]  
- [4, 5] + [2, 3]  
- [6, 7] + [0, 1]  
- [6, 7] + [6, 7]

Figure 3 shows the possible cases of two input values determined by the backward input to “iadd” for the case of ndiv=4, where the range of 32 bit integers are express by −N to N-1. Slant lines in the figure show a relation between x and y to make the same sum. The hatched area means the area of sum of 0 to N2, where N2 means N divided by 2 with truncating the fraction. In this case of ndiv=4, eight square areas which includes hatched portion are the areas to be searched for feasibility.

Figure 4 shows an example of bytecode program including “if” type instruction. The original Java program is

```
int x=15; if(x<=10) x=10; return x
```

![Figure 4. Example of if jump, (a) forward (b) backward.](image)

As for the reverse operation of “if” type instruction, simulation flow must branch at the destination instruction of “if” instruction. We used pseudo instruction “j” as a junction point shown in Figure 4. There is no information of jumped or not when backward control flow reached the “j” instruction. We have to test both of jumped flow and not-jumped flow. If the flow traces jumped flow, it goes into “if” instruction through “o1” port which means “if” instruction was met, otherwise it goes through “o2” port which means “if” condition was not met.

As for the case of “if_icmple”, “if” decision map for the first argument x and the second argument y is shown in Figure 5. If we set ndiv=4, cases to be tested are hatched sections in Figure 5 when the control flow enters through “o2” as an example.

![Figure 5. Decision map of if_icmple for ndiv=4.](image)

4. REDUCTION OF PROCESSING TIME

Already in section 3, we explained that the range processing can be used as shortening method at the sacrifice of resolution. Here, we describe other methods of reducing processing time for our backward simulation.

The first one is to use the result of former simulation which was performed with lower resolution (stepwise we name it). Rejected areas by a simulation of lower resolution correspond to many cases of a simulation of higher resolution, and would be rejected by a simulation of higher resolution. In the implementation, “start” object writes feasible ranges into a file, and “end” object reads them in the following simulation.

![Figure 6. Example of search order from the area near to zero.](image)

The second method is to start the search from expected area of possibility. In many cases, small values compared to the upper limit of integer are used in programs. Therefore, it is better to start test from smaller value such as zero (search from zero, we name it). Figure 6 is an example of search order for the case of ndiv=4.

The third method is to divide the whole value range into nonlinear sections. When possible results are expected to exist in a small area, setting a large area outside of expected area as a single range and rejecting it will swiftly squeeze the area of search. In Figure 7, the whole integer area is divided into four sections of equal division according to log scale (exponential division according to linear scale), which expects possible cases to exist in the area of smaller numbers. The non-negative section needs a special care to add 1 to convert to log scale.
5. IMPLEMENTATION AND BRANCH CONTROL

Here, we describe some more details about our backward simulator implementation. Bytecode objects including pseudo codes ("start", "i", "end") are implemented as an Actor object of Scala Language, which can be executed concurrently by JVM's multithread function.

Messages between simulation objects are expressed by UCF [6] which uses XML style format expressing destination and content. Operand stack of Java is expressed in UCF message and delivered to connected bytecode objects. By this, there is no need for save and restore operand stack for each branch request. In UCF messages, numerical values are expressed by numerical character strings, ranges are comma separated numerical values, character strings are expressed by double quoted strings, and stack is expressed as semicolon separated list with the top item in the first of the sequence. Although UCF message would be long when the stack is deep, it does not seem long for general cases.

As an example, if two ranges of values are [0,127] and [128,255], the message may be,

```
<sim><s>c2</s><i/s>0,127;128,255</i/s><sim>
```

where the destination "sim" is the simulation kernel which controls the whole simulation and transfers all messages, the tag "i" means the source of the message, nesting "s" tag means that the source is port "i" of node "c2", and operand stack has three range pairs. The last range of values (0,255) is intentionally pushed in the stack by "end" to reach "start", which can be used as the range indicating a feasible range of values.

Local variables are defined for each Java method with all of five variable types ( int / long / float / double / reference). At the start of branch operation, local variables are saved into variable stack, and they are restored at the time of new case simulations. Memory references are not supported now, but they would be treated in a similar manner to local variables.

Simulation starts by a trigger from "start" for forward simulation or "end" for backward simulation. Thereafter, "sim" controls the simulation sequence by receiving and responding to messages from simulation objects. Connections between objects are kept in the link table of "sim", and a message from a port of some object will be passed to the connected object through a predetermined port determined by the link table. With such transfer by "sim", forward and backward simulation proceed. The division parameter ndiv is delivered to all of the branch capable objects and used to determine range divisions. It is better to use the same ndiv by all objects because different division size may cause multiple case branches and may greatly increases the number of cases to be tested.

Figure 7. Example of exponential range division.

Figure 8. Branch control of the backward simulator.

The branch control is handled as shown in Figure 8 [7]. Any object which requires case branch determines unique branch ID and sends a "branch request" message with the ID to "sim", which instructs all the objects to save internal status with the branch ID, and starts a simulation of the case specified.

Any object which detected inconsistency sends "nextCase" to "sim", which broadcasts the message to all the objects. Receiving objects reset their status to that of branch start. And the branch request object sends a next simulation case (range of values) if any remains. Otherwise it sends "branchCompleted" to "sim", which will be broadcasted, and receiving objects will return to the status before the current branch.

When a backward control flow reaches "start", it means the range which "end" sent is a feasible range for the program. Therewith, "start" requests next range generation to "end". If no case is remained in "end", "end" sends "branchCompleted" to "sim" and the simulation goes back to the previous branch if exists, otherwise the whole simulation stops.

6. SIMULATION RESULTS

Figure 9 shows a result of backward simulation of Figure 4(b). The lines “Dstrt” indicate the feasible range sent by "end" for the specified ndiv value. By increasing ndiv, the result approaches the exact possible value of 15.

```
ndiv=2  
Dstrt received 0.65534;  
sim elapsed time=0.032 s  
ndiv=2  
Dstrt received 0.724;  
sim elapsed time=0.073 s  
ndiv=2  
Dstrt received 0.14;  
sim elapsed time=0.066 s  
ndiv=2  
Dstrt received 0.14;  
sim elapsed time=0.076 s  
ndiv=2  
Dstrt received 4.14;  
sim elapsed time=0.158 s  
ndiv=2  
Dstrt received 15.18;  
sim elapsed time=2.297 s  
ndiv=2  
Dstrt received 15.18;  
sim elapsed time=2.703 s  
ndiv=2  
Dstrt received 10.14;  
sim elapsed time=8.884 s  
ndiv=2  
Dstrt received 15.15;  
sim elapsed time=13.235 s
```

Figure 9. Output of backward simulation of if simpleconditional.
division. The stepwise simulation shows an excellent reduction effect. The simulation of (c) is better than stepwise, but its performance is dependent to the resulting values. Figure 11 shows three cases of addition, (a) 0+1, (b) 11+1 and (c) 127+1. The factor of up to 10 reveals that starting point is a key to reduce processing time.

We have implemented only 12 out of 205 Java bytecodes, now. About 95 of them including constant data input instructions, single value operations, “i” type operations are easily implementable. Two input instructions other than “add” are thought to take time and need elaboration. Logical instruction may be harder to reduce processing time because they should be handled bit by bit theoretically.

8. CONCLUSION
We showed a concrete method of reverse execution and implemented a backward range simulator for Java bytecode analysis. By this method, it is possible to perform a comprehensive testing of machine coded program with controlling resolution and processing time. Processing time of implemented simulator for simple integer calculation programs was reduced to a practical level by using reducing method of stepwise simulation, search from the expected area and exponential range division. Although there are some difficult machine codes for reverse execution, we hope they can be implemented in a practical simulator by using the concept and methods shown here.

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

10. REFERENCES