A Software Testing Tool with the Function to Restore the State at Program Execution of a Program under Test

Yuhei Otani, Hiroaki Hashiura, and Seiichi Komiya

***Abstract*—** In a software development project, conducting a unit test is an important task but not an easy process. In particular, preparing a unit test procedure for a program that uses external resources such as files and databases as its input tends to be a difficult task since it requires the developer to set up various conditions. This paper proposes a unit test technique that can automatically reproduce runtime conditions of a java program by tracing the execution history of the program for the object generation method and the flow of changes, and also proposes a development support tool. Our technique and support tool allow the developer to automatically generate a unit test procedure for a program for which it is difficult in the past to create a unit test procedure since it uses external resources.

***Keywords*—**Automatic generation, Execution history, JUnit, Unit test

I. Introduction

In a software development project, the software testing process is an important process to prove the validity of software functionality. In the software testing phase, a unit test is conducted to find defects in modules, such as methods and classes that are the smallest unit of software components. Automatically generating unit test cases has become an important task, much research has been conducted[1]-[4]. It is known that a unit test can be done efficiently by using the unit test framework called xUnit. JUnit[5] provided in xUnit is usually used to conduct a unit test of a program written in Java. JUnit provides the framework that can be used to write test cases as a test class and enable automatic execution of test cases.

A test class can be used to generate test cases that are aimed at testing a class or methods of a class and that can be used to represent the test results supposed to be obtained when specific input data is provided. However, it is difficult to create test cases for such a program that uses input data from external resources, such as files or databases, since the behavior of the program may have an impact on the status of the external resources during program execution. In order to conduct a test on such a program, it is necessary to create a test case that does not have an impact on the resource status by reproducing the status of the external resources during program execution. From this viewpoint, there are only few test support tools that can be used to conduct a test for which reproduction of the status during program execution is essential.

This study proposes a unit test technique of automatic test case generation that can generate test cases by reproducing the execution status of a java program based on its execution history. The authors have also developed a tool that supports this technique and evaluated its effectiveness through experiments.
This paper consists of the following sections. Section 2 introduces specific cases in which reproduction of program execution status is essential to conduct a test. Section 3 shows related studies in which test cases are generated based on an execution history. Sections 4 through 6 clarify the specific procedures proposed in this study. Section 7 describes the evaluation experiment conducted using the implemented tool proposed in this study. Section 8 discusses the conclusion and the future direction.

II. What Prompted Us to Start This Study

This section describes the reason why reproduction of the program execution status is essential by showing specific sample cases.

A. A program that uses external resources

Executing a program that uses external resources including file streams, network streams, and database connection has strong impacts on the status of such resources. Therefore, when conducting a test on a program that uses external resources, the user has to develop test cases based on the assumption made for the resource status during program execution. However, typical external resources are unstable and uncertain, and their internal states are highly changeable. As a result, test cases developed based on the assumption of the external resources' states may not represent their actual states on execution. So, it is necessary to set the internal resources to appropriate states on program execution.

Figure 1 shows an expected resource state when a test is executed on the fileProcess method. The fileProcess method

Yuhei Otani is with the Graduate School of Engineering and Science Shibaura Institute of Technology, 3-9-14 Shibaura, Minato-ku Tokyo, 108-8548, Japan (e-mail: ma11038@shibaura-it.ac.jp)

Hiroaki Hashiura is with the Faculty of Information Sciences and Arts, Toyo University, 2100 Kujirai, Kawagoe-shi Saitama, 350-8585 Japan

Seiichi Komiya is with the Graduate School of Engineering and Science Shibaura Institute of Technology, 3-9-14 Shibaura, Minato-ku Tokyo, 108-8548, Japan

takes an object of the FileInputStream class as its argument, uses this object to read the contents of a file, and returns the results generated by processing the read values. This method expects, as the prerequisite, that it read the value "D" on Line4 in file1. To satisfy this condition, the argument in of the fileProcess method should be the input stream associated with file1 and the cursor should point to Line4. However, since the cursor points at the beginning of the file, Line1, when the stream is created from the file, a read operation is allowed to start only from Line1. To satisfy the prerequisite, the internal state of the stream must be set to allow the read operation to start from Line4.

One of the ways to set up the internal state is to capture the internal state of a running stream object and reproduce it. Serialization is usually used as a typical way to reproduce the internal state of a running object. Serialization is a technique to convert the internal states of an object being handled in a program into byte strings or something represented in the XML format and save them in a file. However, serialization cannot be adopted for some objects associated with external resources such as input and output streams, since it is not possible to serialize these objects. To reproduce the object states without serialization, it is necessary to find the object states during program execution and set up the environment for operations. For example, the read method of the FileInputStream object can be repeatedly called until the cursor points to Line4 to set the stream object to the expected state. However, usually it is not easy to implement such a setup procedure.

B. Programs with complex dependency relationship

To write a test class for a class, it is necessary to instantiate the targeted test class. At this stage, if the class to be tested has a dependency relationship, it may be quite difficult to instantiate the class. The dependency relationship, in this case, represents the relationship that requires an object from another class to instantiate the targeted class. An increasing number of setup procedures are required to resolve the dependency relationships as the number of objects required for the arguments of the constructor increases or the complexity for generating the objects increases.

Figure 2 shows several examples for a class of which dependency relationship can be easily resolved and for another class of which dependency relationship cannot be easily resolved. The example of classA represents a class of which dependency relationship can be easily resolved since it can be instantiated only by passing a value of int type to the argument of the constructor. On the other hand, classB can be instantiated only if an object of the class that is equipped with Connection which represents a specific connection with a specific database is passed. In this case, it is difficult to resolve the dependency relationship since the object cannot be prepared unless an actual connection is implemented with the database. In the case of classC, it is necessary to pass objects D, E, and F of other classes as its arguments to instantiate it. Probably, the D, E, and F classes may have further complex dependency relationships. As the number of dependency relationships increases, it becomes more difficult to run the class and develop test cases. Reproducing the object states during program execution makes it needless to resolve such dependency relationships.

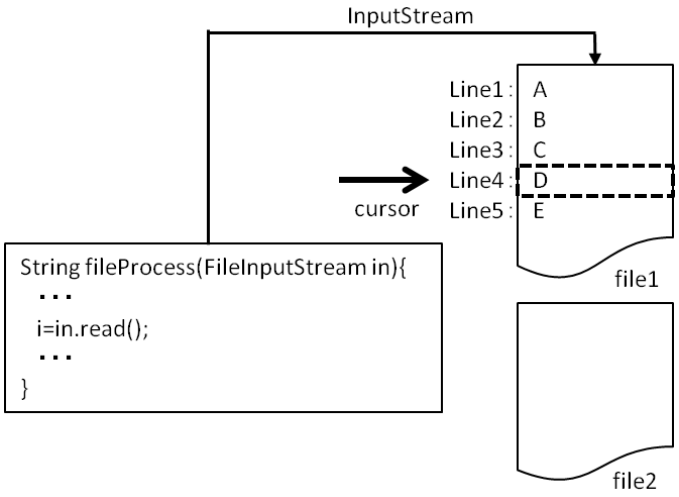

Fig.1 Expected stream state.

```
classA {
  A (int num) {
    //Constructor
  }
}
```

```
classB {
  B (Connection con) {
    //Constructor
  }
}
```

```
classC {
  C (D d ,E e ,F f) {
    //Constructor
  }
}
```

Fig.2 Classes with dependencies.

III. Related Studies

This section describes a survey of related works in which unit tests are generated from execution results and compares those works with our study.

Elbaum et al. proposed a unit test generation technique with which independent unit tests are generated for each method by classifying execution results into several groups and independently rerun each method on each group[6]. The following discusses how to implement this technique. First, use XStream to save the internal states of the program before and after a method is executed during a system test. Then, run a method of which saved preconditions are restored. This technique finally compares the post-conditions generated after

the restored method is executed with the post-conditions saved during the system test to check to see if they are identical. XStream provides a library that can be used to serialize the states during program execution, writes them to a file after converted to the XML format, and restore them. This technique cannot be used to restore the objects that cannot be serialized. Another feature of this tool is to increase the execution efficiency by finding the objects that can be referenced from a specific method and save only the information specific to the method. Our study aims at capturing and restoring the execution states more completely. As a result, our technique is less efficient than Elbaum's technique from the viewpoint of stored data volume. On the other hand, it can handle automatic generation of unit tests for programs that cannot be serialized and to which Elbaum's technique is not applicable.

IV. Our Technique Proposed in This Study

To cope with the issues described in Section 2, the authors proposes a technique that can be used to automatically generate unit tests by capturing an execution history generated by test data and composing the contents of the execution history. The execution history can be used to clarify the operations performed by each object during program execution. Our technique traces the operations on external resources executed by stream objects and reproduces the states of streams by executing the traced operations using the test classes. In addition, since the execution history can be use to clarify how the objects were generated, the dependency relationships can be resolved by generating the objects according to the execution history exactly. The following describes the actual steps performed with our technique.

Step 1. Develop test cases manually to perform the entire program based on the requirements specification.

Step 2. Acquire the execution history by running the test cases created in Step 1.

Step 3. Analyze the execution history and automatically generate a unit test program.

First, the system requirements specification is analyzed to clarify the program behavior and the test data is defined as input items to the program. Since the unit tests are generated from the execution history acquired by running the program with the test data defined here, the test data should be prepared as the one that is related to the state to be reproduced.

Then, the program is executed with the defined test data to acquire the execution history. The execution history is acquired with traceglasses discussed later in this paper. As shown in Figure 3, the execution history can be used to trace the input and output of each method as well as to identify the impacts on a specific object made by the method.

The acquired execution history is analyzed to generate a unit test for each method by reproducing the same states as the ones observed on program execution for each method that appears in the execution history. Section 6 describes the technique to analyze the execution history. For our technique, a specific environment is required to execute a program and acquire its execution history. It is necessary for our technique to work properly that the requirements specification is prepared and the program has been proved to start successfully and run with no bugs.

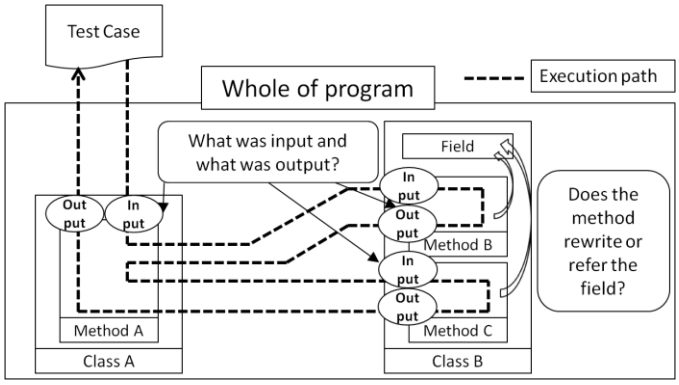

Fig.3 Information gained from the execution history.

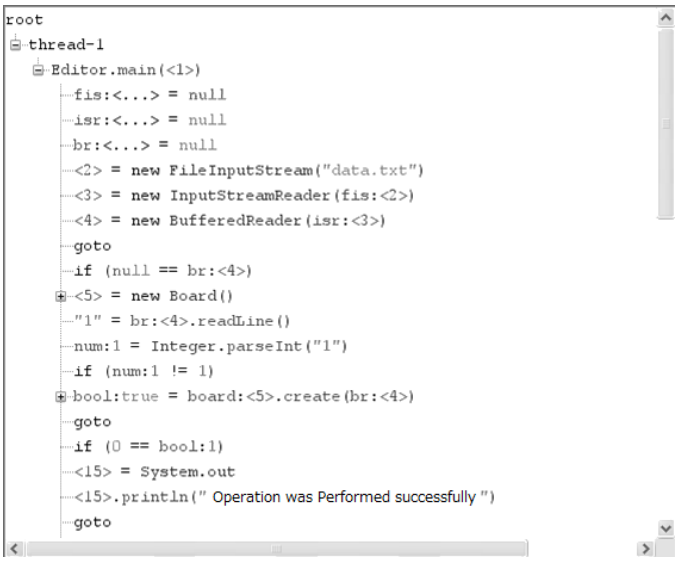

Fig.4 The viewing area of traceglasses.

V. Tools Used in Our Technique

Our technique uses traceglasses as the tool to acquire the execution history. It generates its output and test cases as a test class of Junit. The following describes these tools.

A. *traceglasses[7]*

Traceglasses is a debugger that can collect log data as trace records during execution of a java program and allow the user to track defects interactively. It allows the user to track object generation incidents and all operations since it keeps trace records of executed programs as much as possible from the beginning to the end.

Figure 4 shows an actual display screen of traceglasses. Traceglasses gives a unique ID such as <1> to identify each object it has used. The operations performed by an object can be tracked by searching for the corresponding ID. For example, the trace line of

bool:true = board:<5>.create(br:<4>)

indicates that the method create is called from the object <5>

stored in the variable board, the object <4> stored in the variable br is specified as the argument to the method, and the return value of true is stored in the variable bool. That is, the corresponding instruction line executed in the source code is bool = board.create(br);. The values stored in the objects <4> and <5> can be identified by searching the preceding lines located before this instruction line.

For example, the object <5> is found to be an object of the Board class since it is generated in the trace line of <5> = new Board().

B. JUnit

JUnit is a framework that works as a test driver and has been developed to automate unit tests of java programs. With JUnit, unit tests are written as a test class. The test flow using a test class is as follows:

1. Set up the method to be tested.
2. Use assertions to examine the preconditions of the method.
3. Run the method.
4. Use assertions to examine the return values of the method, if any.
5. Use assertions to examine the post-conditions of the method.

In this case, the setup procedure includes generating objects required to call the method to be tested and setting each object to a value that satisfies the test conditions. An assertion is a method, such as assertEquals and assertTrue, which can be used to determine if the test has succeeded or not. The assertEquals method takes two values as its arguments and determines that the test has succeeded when they are identical and that the test has failed when they are not identical.

VI. Analysis of Execution History

This section discusses how to compose the contents of the execution history to generate a test class. Figure 5 shows the entire analysis flow.

A. Acquiring the method information invoked during execution (I)

First, the information of the method to be tested is acquired to generate a test class. Our technique deals with testing of the methods that are invoked during execution, except for those in a library such as the standard library. At this stage, the following points are examined for each method invoked.

1. The ID of the object that called the method
2. The arguments to the method
3. The return value from the method
4. The names of the fields referenced by the method called from the object
5. The ID of the object (*) changes are made in the method
6. The field name of (*)
7. The value of (*) before change
8. The value of (*) after change

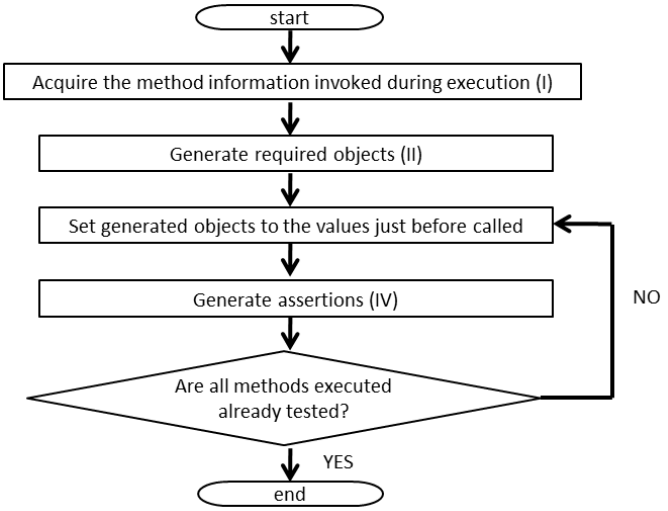

Fig.5 The flow chart to show the steps to analyze the execution history.

```
1 | bool:true = board:<5>.move(br:<4>) . . .
2 | <6> = this:<5>.al . . .
3 | <14> = <6>.get(num:0) . . .
4 | rect:<14>.move(dx:1,dy:1) . . .
5 |   this.<14>.x=2
6 |   this.<14>.y=2
```

Fig.6 Example of trace to obtain information of method.

Figure 6 shows the trace lines required to collect information for the move method when it is selected as the test target. This trace shows that the ID of the object that called the method is 5, the argument of the method is the object of ID4, the return value from the method is true, the name of the field referenced from the object (ID5) that called the method is al, the ID of the object changes are made in the method is 14, the names of its fields are x and y, and the value after change is 2. The value before change is determined by searching the trace lines before the method is called and set to the value found first, which is the value of the object before the method is executed. As described above, the object information required to call the test target method and the side effect information of the method required to generate assertions is acquired by searching the preceding and following lines around the line of the test target method.

B. Generate required objects (II)

To execute the test target method, the object that calls the method, the object passed as the argument to the method, and the objects required to generate these objects are required. These objects are generated according to the steps described in Figure 7.

In Figure 8, when the method move on Line 5 is selected as the test target, the objects required for execution are the objects ID5 and ID4. These two objects are searched for backward starting from the line on which the test target method is called. Then, it is found that ID5 is generated on Line 4 and ID4 is generated on Line 3. Since the object ID3 is required to generate the object ID4, the line on which the object ID3 is generated is

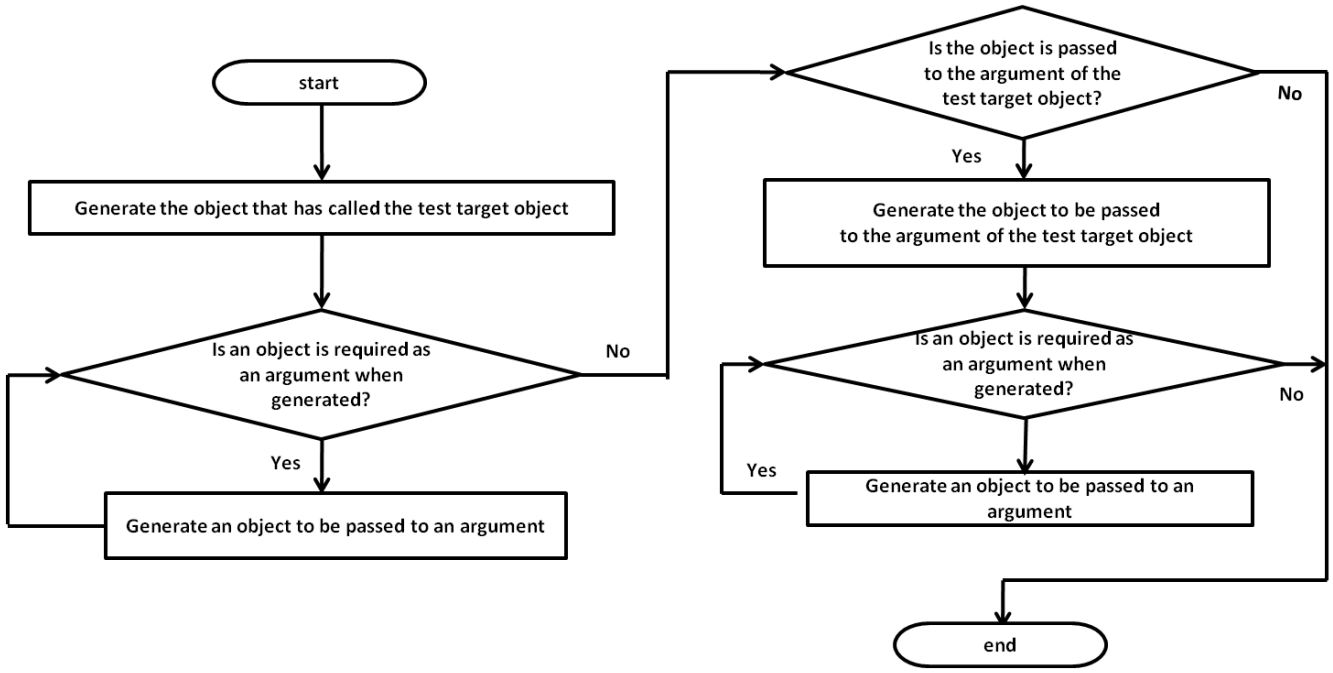

Fig.7 The flow chart to show the steps to create an object.

```
1 <2> = new FileInputStream("data.txt")
2 <3> = new InputStreamReader(fis:<2>)
3 <4> = new BufferedReader(isr:<3>) . . .
4 <5> = new Board() . . .
5 bool:true = board:<5>.move(br:<4>)
```

Fig.8 An example of trace to generate necessary objects.

```
1 <5> = new Board() . . .
2 <14> = new Rectangle(1,1,2,3) . . .
3 <6> = <5>.al
4 true = <6>.add(rect:<14>) . . .
5 bool:false = board:<5>.create(br:<4>)
```

Fig.9 An example of trace to reproduce related objects.

searched for by backtracking the lines beginning from the line on which it is used.

It is found that the object ID3 is generated on Line 2. As just described, when an object is required to generate another object, it is searched for recursively. Once the way to generate the object is determined, a set of code lines to generate the object is generated according to one of the following patterns.

- The new operator is used to generate the object (except for arrays):
  - Type name $+Object ID = new Type name(Variable name);
- The new operator is used to generate an array:
  - Type name[] $+Object ID = new Type name[Number of data items];
- The return value of the library is used to generate a new object:
  - Type name $+Object ID = $+Object ID.Method name(Argument);

C. Set generated objects to the values just before called(III)

After an object is generated, it must be restored to the same state as the runtime state. To reproduce the state of the object, all method calls and field changes executed between the period from object generation to actual use must be re-executed.

The following describes how to reproduce the state of the object ID5 to its actual execution state. Since ID5 is generated on Line 1, it is necessary to search for the operations of ID5 executed between Line 1 and 5. It is found that the al field of ID5 is called on Line 3. On this line, the filed al of ID5 is stored in ID6. It means that, at this point, the field al in ID6 is the same as that in ID5. Hereafter, since the operations using the field al of ID6 have the same effects as the operations using that of ID5, it is necessary to track the operations performed by ID6. Since ID6 calls the method add on Line 4, it is necessary to generate a code line to call add to reproduce the actual execution state. In addition, since a new object ID14 appears as an argument of add, this object must be generated according to the procedure of II and its execution state must be reproduced according to the procedure of III.

D. Generate assertions (IV)

The setup task to execute the method is completed by the procedures II and III. Next, the code lines to examine the precondition and the post-condition of the method need to be generated based on the information acquired in (I) following the procedure shown in Figure 10. The assertEquals method is used for each assertion.

Generation of a unit test for a method completes through the procedures from II to IV. Perform the procedures on all the methods executed when the test case is performed until all of the unit tests are generated.

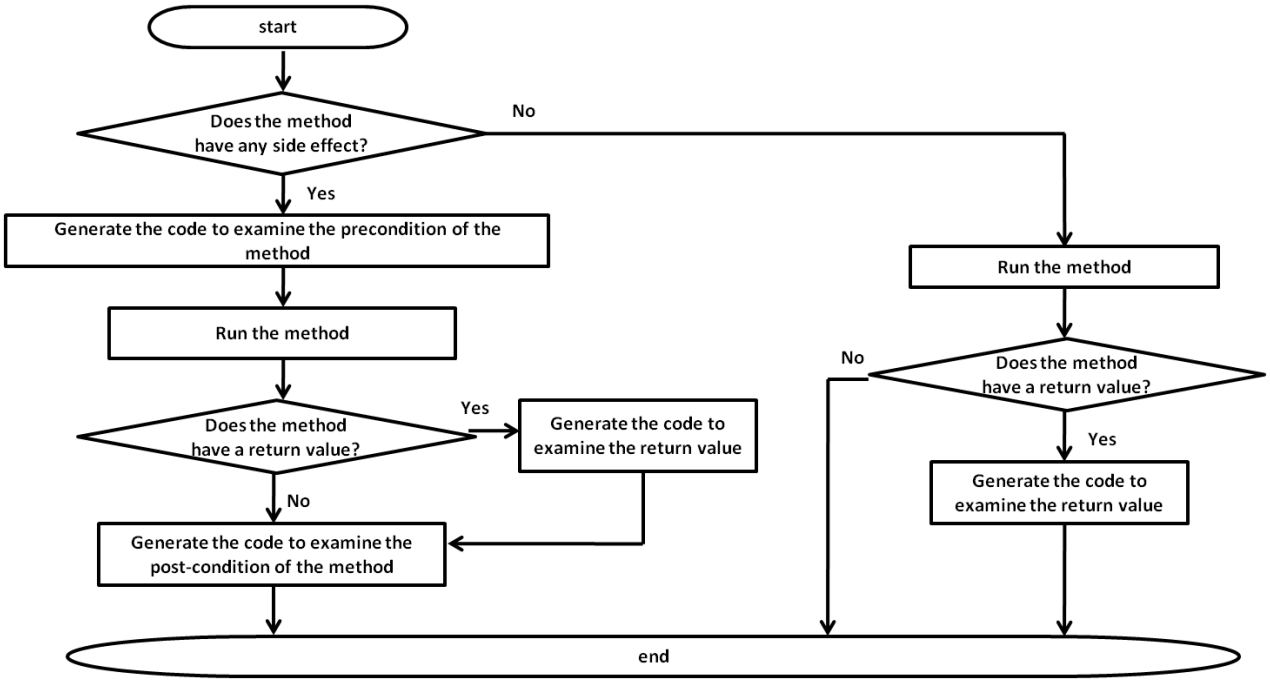

Fig.10 The flow chart to show the steps to generate assertions.

```
1  public void test(){
2    java.io.FileInputStream = $2 = new
           java.io.FileInputStream( "data.txt" );
3    java.io.InputStreamReader $3 = new
           java.io.InputStreamReader($2);
4    java.io.BufferedReader $4 = new
           java.io.BufferedReader($3);
5    editor.Board $5 = new Board();
6    $4.readLine();
7    assertEquals(true,$5.create($4));
8  }
```

Fig.11 Generated unit test program.

E. Test cases to be generated

A unit test program as shown in Figure 11 is generated through the procedure listed in Figure 11. Lines from 2 to 5 are generated by the procedure of II, Line 6 is generated by III, and Line 7 is generated by IV. There are some problems in this program such that no exception processing is provided and restrictions are imposed on access. For example, the readLine method on Line 6 requires exception handling and the create method called on Line 7 may be a private method.

We use javassist[8] to resolve these problems. Javassist is a library that can be used to examine the definitions of class files and change their contents. Javassist is used to minutely examine each constructor and method called from the generated test program for the type of Exceptions. If it is found that exception handling is required, throws is used to generate an operation to generate an exception.

As for the problem of access restriction, javasisst is used for the original source file to change its attribute to public. By using the changed class file, it is possible for the test program to run even though the access restriction is set to private in the original source file. Since the original source file is not changed, recompiling the source files after the test is completed can restore the access restriction to the original state.

VII. Experimental Evaluation

The authors adopted the following benchmarks to evaluate the effectiveness of the test cases generated by the system to implement our technique.

- Test success rate = Number of passed test cases / Number of generated test cases
- Suitability rate = Number of test cases in which the execution state is properly reproduced / Number of generated test cases
- Instruction coverage rate = Number of instructions executed / Total number of instructions
- Branch coverage rate = Number of branches executed / Total number of branches

A. Program used in the experiment

The authors developed a small program tested in the experiment. According to the purpose of this study, this program contains the following method that is unable to be serialized.

public boolean create(BufferedReader br)

Table 1 shows the size of the program. The authors used this program to produce test cases by providing test data that triggers execution of all defined methods and execution of each instruction as much as possible.

B. Results

Table 2 shows if the generated test cases can be used to successfully test the system. Table 3 shows the coverage rate of test cases against the entire code lines.

C. Consideration

Table 2 shows that 492 test cases were generated, and all of them passed the test and successfully reproduced the execution states. The suitability rate is 100% so that the experiment successfully proved that the test cases generated by our technique could properly reproduce the execution states. In addition, it proved that no test case that led to a failure was generated. Next, as for the Rectangle class, Table 3 shows that all instructions and branch conditions in the source code are covered. On the other hand, as for the Board class, the instruction coverage rate is 91% and six instructions are not covered while the branch coverage rate is 100%. Although it is desirable that the coverage rate is 100%, it is not necessary to be 100% since a program may contain some paths that are not expected to be executed. Examining the code lines for not executed instructions clarified that the instructions related to exception handling of IOException are not covered as shown in Figure 12. This exception is not generated except for a special situation such that the stream has been closed before the readLine method is called. If the stream is always open according to the program specification when the try statement in Figure 12 is invoked, since no test case can be automatically generated for these instructions, it is necessary to develop test cases manually if testing of such statements is mandatory. If there is a path in the program in which the stream is closed when the try statement is invoked, it is considered that the test case for such instructions can be generated by running the system with input data that walks through such a path.

Table 1. The size of the experimental program.

Count＼Class name	Board	Rectangle
Number of methods	5	8
Number of lines	113	49

Table 2. The number of generated test cases and their contents.

Number of generated test cases	492
Number of succeeded test cases	492
Number of test cases with which the execution states are reproduced	492
Test success rate	492/492(100%)
Suitability rate	492/492(100%)

Table 3. Code coverage.

Number of benchmarks＼ Class name	Board	Rectangle
Instruction coverage rate	59/65(91%)	17/17(100%)
Branch coverage rate	34/34(100%)	8/8(100%)

```
1 try{
2   str=br.readLine().split(“,”);
3 }catch(IOException e){  //Not executed
4   e.printStackTrace();     //Not executed
5   return bool;       //Not executed
6 }
```

Fig. 12 Instructions that were not tested.

VIII. Conclusion and Future Direction

This paper discussed generation of test cases that reproduce the execution states of a program to cope with the difficulty of program test that handles external resources such as files. To resolve such a problem, the authors have developed a tool that automatically generates unit tests by capturing the execution history of a program and composing the contents as unit tests.

It is considered that the unit tests generated by this tool can be applied to a system composed of untested programs. An defect might be found at an unexpected location in an untested program when its code is altered. To cope with such a defect, it is effective to develop test cases that can be used to validate current code behaviors before the code is altered[9]. It is effective to use our tool for untested programs since our technique acquires current program behaviors as a program history and converts it to test cases. On the other hand it is not suitable to use for finding new defects.

Two of the currently identified problems of this tool are that it takes longer execution time and more than one test case with the same value is generated as the program size grows since it generates test cases for all methods executed. As the future direction of our study, we plan to improve the tool by developing an algorithm to eliminate test cases with the same value and allow the user to specify the methods to be tested. In addition, to improve the effectiveness of our technique, we plan to conduct evaluation experiments that apply our technique to the programs that handle input from a database, in which testing is more difficult than input from files.

References

[1] A.Nadeem, M.Jaffar-Ur-Rehman, Automated test case generation from IFAD VDM++ specifications, SEPADS'05 Proceedings of the 4th WSEAS International Conference on Software Engineering, Parallel & Distributed Systems, No.28

[2] M.Alshraideh, Using program specific search operators in test data generation, ECC'10 Proceedings of the 4th conference on European computing conference, pp.132-138

[3] S.Dhawan, K.S.Handa, R.Kumar, Software testing: perception on exploration and ad-libbing, MACMESE'09 Proceedings of the 11th WSEAS international conference on Mathematical and computational methods in science and engineering, pp.113-118

[4] D.Caprita, V.Mazilescu, Automated Software Testing for PHP Web Based Applications, Sustainability in Science Engineering Volume II. p.285. WSEAS International Conference. Proceedings. Mathematics and Computers in Science and Engineering. Vol.2, No.11

[5] JUnit, http://www.junit.org/

[6] S.Elbaum, H.N.Chin, M.B.Dwyer, and M.Jorde, Carving and Replaying Differential Unit Test Cases from System Test Cases, IEEE Transactions on Software Engineering, Vol.35, No.1, pp.29-45, January 2009

[7] K.Sakurai, H.Masuhara, and S.Komiya, Traceglasses: A Trace-based Debugger for Realizing Efficient Navigation, IPSJ Special Interest Group on Programming, Vol.3, No.3, pp.1-17, June 2010

[8] S.Chiba and M.Tatsubori, Structural Reflection by Java Bytecode Instrumentation(Special Issue on Groupware of the 21st Century), Transactions of Information Processing Society of Japan, Vol.42, No.11, pp.2752-2760, November 2001
[9] M.C.Feathers, Working Effectively With Legacy Code, Prentice Hall, 2004

Yuhei Otani He received a bachelor's degree in engineering from Shibaura Institute of Technology, Japan in 2011. At present, He is enrolled in postgraduate of the Shibaura Institute of Technology.

Hiroaki Hashiura He received a bachelor's degree in engineering from Shibaura Institute of Technology, Japan in 2002. He received a professional degree in engineering management from Graduate School of Engineering Management, Shibaura Institute of Technology, Japan in 2005. He received Doctorate in engineering from Graduate School of Engineering, Shibaura Institute of Technology, Japan, in 2008. At present, He is an assistant professor of Toyo University.

Seiichi Komiya He received the degree of the B. S(C), from Saitama University, Japan, 1969. He received Dr. Eng. from Shinshu University in March 2000. He was been working for Hitachi Ltd. as a software engineer during 1970-2001, and has been on loan from Hitachi Ltd. to Information-technology Promotion Agency Japan (IPA), a substructure of MITI, during 1984-1999. He has studied the frameworks to construct many kinds of CASE tools (e.g. an automatic programming system, a software collaborative distributed development environment, etc.), software specification/design process, CAI and Intelligent CAI at IPA. In IPA, he has been a principal researcher of Software Technology Center during 1988-1999, and was also an assistant to director general of Laboratory for New Software Architectures during 1991-1998. He works for Shibaura Institute of Technology as a full-time professor since April 2001. He was also a visiting professor of Tokushima University in 1993. He was also a visiting lecturer of Chiba University during 1995- 2009, and a visiting professor of a graduate school of Shibaura Institute of Technology 1997-2001. He was a manager/a vice-chairman/a chairman of SIG-KBSE/IEICE during 1992-1994/1994/1996/1996-1998. He was also a member/a manager of editorial committee of IEICE transactions 1994-1999/1998-1999. He was given a title of “IEICE fellow” from IEICE in 2010.