A Model Based Testing Approach for Java Bytecode Programs

Safaa Achour*, Mohammed Benattou
LASTID Laboratory, Ibn Tofail University, Kénitra, Morocco.

* Corresponding author. Tel: 0664 707 560; email: safaa.achour@uit.ac.ma
Manuscript submitted May 10, 2018; accepted July 8, 2018.
doi: 10.17706/jcp.13.9.1098-1114

Abstract: In this paper, we propose an approach to detect non-conformance between a given Java Bytecode program and its specification in the context of unit testing. The main goal of the proposed work is to use the specification pre-state, i.e. the invariant and the precondition, to guide the generation of the test data and the post-state, i.e. invariant and post-condition, as test oracle. In other hand, the code structure makes it possible to know the execution paths of testing method that not be conform to its specification. However, the specification and the System Under Test are not at the same level of abstraction. In the sense, we propose to express the specification at the Bytecode level using static Bytecode instrumentation.

Key words: Java Bytecode, model-based testing, static Bytecode instrumentation, constraint model.

1. Introduction

Tests and proof are the principal approaches for software verification. Several methods were developed to insure the correctness of a complex system using proof based on formal methods, either to simplify the mathematically burdensome process or to optimize the computationally complex task of model checking [1]. If we consider that the verification process allows guaranteeing the absence of certain classes of errors, the reliability of the verification results depends on the distance between the abstraction and the real application. In other hand, tests help to stimulate the behavior of software by applying inputs and checking if the specifications are respected in the output, but its reliability depends on the count of test oracles and the efficiency of the input data to parse the maximum states of the application under test [2]. Even if the testing process is considered very expensive and difficult to be implemented in the industrial context, we believe that the model based testing can obtain high confidence for validating an implementation.

In this context, constraint-based testing introduced by Offutt in 1991 [3], for generating test inputs. Several techniques were developed to achieve this goal, in particular those who use the Symbolic Execution due to its ability to generate a high-coverage test suite and find deep errors on complex software applications [4] and other combining a symbolic execution with dynamic constraint solving.

However, often of these techniques are restricted to source code level programs, while for many applications one needs to be able to also verify the executable code, i.e. Java Bytecode. Different possible reasons for this exist: Java Bytecode program can have bugs since the methods used for Java software testing does not necessarily remove all possible bugs from its source program. Furthermore, the source code of an application is not always available; and even this is the case, structural testing requirement can still be derived and used to assess the quality of a given test set [5]. On the other hand, the Bytecode is

already free of compilation errors and optimized for execution. So, we think that, the testing methods for Java applications at the Bytecode level are necessary. In Java testing context, we have the assurance that if we dispose for each method of each class of its specification and its Bytecode, we can firstly detect the invalid execution errors, and we can secondly perform code coverage of the testing method under test.

Several works have adapted structural testing techniques on program at the Bytecode level. These works include extracting a control flow graph from Bytecode program [6], [7], performing symbolic execution of Bytecode [8], or using constraint based techniques to generate test inputs from java Bytecode programs [9], [10]. However, few of them have been interested in testing the behaviour of the Bytecode program with respect to its specification. The main purpose of our testing approach is to extract testing information from Java Bytecode program and its functional specification expressed in pre/post conditions and invariant. As it known, the System Under Test (SUT) and the user specification are not at the same level. The program is at Bytecode level whereas the specification is in high-level abstraction. In Java software context, we attempt to respond to the two main questions: how can we express the specifications for the Java Bytecode, and how we can generate test data to detect the invalid execution paths of the target application that not be conform to the user specification.

In this paper, we propose to exploit the information given by the user specification and contrary to [9] where the authors generate input test data using the constraint memory model and the test input reach only selected location in the byte code program, our approach aims to detect non-conformance between a given Bytecode program and its specification in the context of unit testing. The main goal of the proposed work is to use the specification pre-state, i.e. the invariant and the precondition to guide the generation of the test data and the post-state, i.e. invariant and postcondition, as test oracle. However, the specification and the System Under Test are not at the same level of abstraction. In this sense, we propose to express the specification at the Bytecode level using static Bytecode instrumentation.

This paper is organized as follow: Section 2 presents a related work of the model based testing and its application at Bytecode level, Section 3 gives a brief description of Constraint Memory Model representing Java Virtual Machine (JVM) states, Section 4 presents our proposed testing method, Section 5 illustrates by the given example how the non-conformance of execution paths can be detected, finally Section 6 gives some concluding remarks and outline our future works.

2. Related Work

2.1. Model Based Testing

Testing is an essential activity in software engineering. In the simplest terms, it amounts to observing the execution of a software system to validate whether it behaves as intended and identify potential malfunctions [11]. In this context, Model-Based Testing (MBT) has become an efficient way for validating an implementation. While the program is being developed, based on informal requirements, the formal model is written, validated and verified. Tests are derived from the model and run on the System Under Test (SUT).

In object oriented modeling, a formal specification defines operations by collections of equivalence relations and is often used to constrain class and type, to define the constraints on the system states (invariant), to describe the pre- and post-conditions on operations and methods, and to give constraints of navigation in a class diagram [12]. Various techniques use constraint solving techniques with annotated programs either to generate test cases or to verify program correctness [13]. However, tests reliability depends on the count of test oracles and the efficiency of the input data to parse the maximum states of the System Under Test SUT. In this context, constraint-based testing introduced by Offutt in 1991 [3], combines a symbolic execution and dynamic constraint solving [14] in order to generate test inputs.

There are several approaches to automatic test data generation based on formal specifications. In [15], they propose an approach for generating test data based on OCL constraints using partition analysis of individual methods of class. The set of given constraints are reduced using the mathematical Disjunctive Normal Forms. The work presented in [16], propose an automated random testing method as a practical tool to assure the correctness of interface specifications. In [17], the authors presented a method based on automated test generation from B models using Constraint Logic Programming. They compute boundary goals and states using a specific solver to build test cases by traversing the constrained reachability graph of the specification. They have applied their technique and tool on the GSM 11.11 specification.

2.2. Java Bytecode Testing

Several works have adapted structural testing techniques on program at the Bytecode level. These works include extracting a Control Flow Graph from Bytecode program, performing symbolic execution of Bytecode, or using constraint based techniques to generate test inputs from java Bytecode programs. In [7], the authors show how the general control flow graph can be generated from a given java card Bytecode program extracted from the CAP file. In [18], they describe a coverage testing tool named JABUTI, designed to test Java programs and Java-based components. The proposed tool extracts from the java Bytecode the intra-method control-flow and data-flow testing requirements used to generate or assess the quality of a given test set. In [8], the paper presents Symbolic PathFinder (SPF); a software analysis tool that combines symbolic execution with model checking for automated test case generation and error detection in Java Bytecode programs. The authors present in [19] a new rule-based testing (RBT) approach to automated generation of test inputs from Java Bytecode without using fitness functions.

In [9], Charreteur and Gotlieb propose a new automatic white box test inputs generation for Java Bytecode Programs, based on constraint programming that aim at building an input state of the Java Virtual Machine that can drive program execution towards a given location within the Bytecode. This technique allows, through constraint programming, to restrict the number of input data to be labeled in order to cover the sequence of instructions that leads to the test objective. We can distinguish two principal contributions in the proposed works: firstly the authors perform backward exploration at the Bytecode level; and secondly they propose a new constraint-based model of the JVM defined with the notion of constrained memory variable. They implement their approach in a tool called JAUT that can generate input memory states for reaching specific location within Java Bytecode programs.

The main purpose of the proposed testing approach is not only to extract the testing information from given specification but also to explore the memory constraint model deduced from Java Bytecode program [9], [10]. Indeed, the authors generate data test inputs using only the constraint memory generated from Bytecode program. The main goal of their proposed work is to deal with the problem of the reachability of an instruction in a Java Bytecode program.

In our work, we consider both Java Bytecode Program for extraction of the program structure, the code coverage, and the user specifications expressed in form of precondition, post-condition and class invariant, that represent the invocation context of the methods. We specify the program at the Bytecode level using static Bytecode instrumentation. Our approach aims to generate valid input test data relatively to the method precondition and class invariant, and to detect the non-conformance between a given Bytecode program and its specification in the context of unit testing.

3. Constraint Memory Model

This section gives a brief description of the Java Virtual Machine (JVM) representation and the existing memory constraint model. The memory model [9], [10] uses Constrained Memory Variables (CMV) to represent JVM states.

The JVM states represent runtime data storage locations such as registers, operand stacks and heap data. The registers are used to store the parameters and the local variables of a method. When the method is dynamic the first register contains the reference to the object (this) that calls the method. The operand stack is used to perform the calculations of the method whereas the heap is the area of memory used by the JVM for dynamic memory allocation. The Fig. 1 shows an example of Java Bytecode method execution.

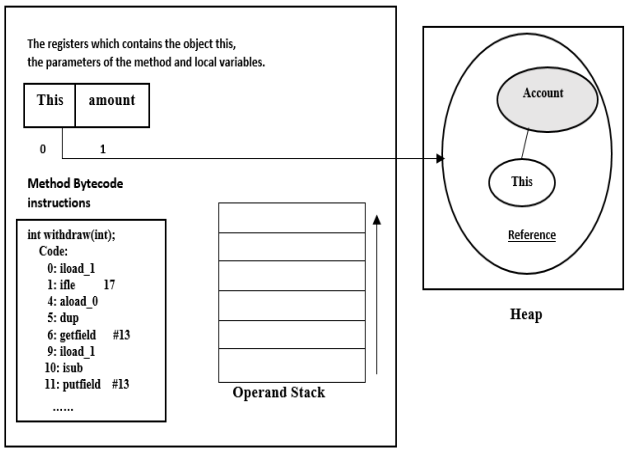

Fig. 1. Example of Java Bytecode method execution in the JVM.

The modelling by constraints of Java Bytecode has required the definition of memory model [9], [10] where a memory state is defined as the state of registers, the state of the stack, and the state of the heap. This Memory Model is based on the notion of constrained memory variables (CMV) which are used to represent JVM states.

```
Public class Account {

private int balance;

public Account(int balance){

this.balance = balance; }


public void withdraw(int amount){

 if(amount > 0 && amount < balance/2)
   balance = balance - amount;
 else if(amount < 0)
   balance = balance - amount * 15/100
 else
   balance = balance - amount * 25/100;
     }
     //.....
```

Fig. 2. Example in Java source code.

A CMV contains data storage locations where data can be represented by variable along the domain. As it represented formally in [9] the CMV M is a tuple (F,S,H) where F denotes the set of registers, S the operand stack and H denotes the heap.

```
public void withdraw(int);
      Code:
          0: iload_1
          1: ifle              27
          4: iload_1
          5: aload_0
          6: getfield          #13              // Field balance:I
```

```
 9: iconst_2
10: idiv
11: if_icmpge      27
14: aload_0
15: dup
16: getfield       #13          // Field balance:I
19: iload_1
20: isub
21: putfield       #13          // Field balance:I
24: goto           66
27: iload_1
28: ifge           50
31: aload_0
32: dup
33: getfield       #13          // Field balance:I
36: iload_1
37: bipush         15
39: imul
40: bipush         100
42: idiv
43: isub
44: putfield       #13          // Field balance:I
47: goto           66
50: aload_0
51: dup
52: getfield       #13          // Field balance:I
55: iload_1
56: bipush         25
58: imul
59: bipush         100
61: idiv
62: isub
63: putfield       #13          // Field balance:I
66: return
```

Fig. 3. Example in Bytecode (of the withdraw original method).

Each Java Bytecode instruction of the program is seen as relation between two memory states: before and after the execution of this instruction. Indeed, each Java Bytecode is seen as a relation among two CMVs: the CMV Mj before the activation of Bytecode and the CMV Mk after its activation and before the activation of the following Bytecode in the considered sequence of instructions.

The tuple (F,S,H) contains variables and domains. Integer and references are modelled by Finite domain variables (VTPR designing Variable of Primitive or Reference Type). Their default variation domain depends on the size of their type. The default domain depends on the size of their type. The default domain of a reference can point to every object of the heap; the null value can also be part of the domain.

In other hand, objects of the heap are modelled by pair elements; the first one is the type variable that represents the class of the object and the second element is a mapping associating an integer or reference variable to each attribute, which correspond to the value of the attribute.

In a CMV, the registers are modelled by function that associates a VTPR (the value contained in the register) to an index i, the operand stack is modelled by a sequence of VTPR in which its first element is considered as its top. As to the heap, it corresponds to a mapping from a set of addresses to a set of objects.

Consider the Java program of Fig. 2 that implements the class Account. The Bytecode program shown in the Fig. 3 correspond to the method *withdraw()*. In order to illustrate how a memory constraint model can be generated form an execution path of a byte code corresponding to a given method, we show in the following by an example the constraint memory model of the path [0 – 1- 27- 28 – 31- 32- 33- 36- 37- 39- 40- 42- 43- 44- 47-66] of the method withdraw().

We start with the initial state: $M_{init} = (F_0, \mathcal{E}, H_0)$, *and* $F_0 = \{0 \rightarrow This_r, 1 \rightarrow amount_i\}$. Note that the symbol Ɛ designates the empty stack.

0: iload_1 1: ifle 27 27: iload_1 28: ifge 50 31: aload_0 32: dup 33: getfield #13 36: iload_1 37: bipush 15 39: imul 40: bipush 100 42: idiv 43: isub 44: putfield #13 47: goto 66 66: return	$CMV_0 = (F_0 , amount_i , H_0)$ $CMV_1 = (F_0, , \mathcal{E}, H_0)$, **$amount_i < 0$** $CMV_{27} = (F_0, amount_i , H_0)$ $CMV_{28} = (F_0, \mathcal{E}, H_0)$, **$amount_i < 0$** $CMV_{31} = (F_0, This_r , H_0)$, $CMV_{32} = (F_0, This_r.This_r , H_0)$, $CMV_{33} = (F_0, balance_i.This_r , H_0)$, **$This_r \neq null$, $(This_r , balance_i) \in H$,** $CMV_{36} = (F_0, amount_i.balance_i.This_r , H_0)$ $CMV_{37} = (F_0, 15.amount_i.balance_i.This_r , H_0)$ $CMV_{39} = (F_0, MUL_i.balance_i.This_r , H_0)$, **$MUL_i = 15 * amount_i$** $CMV_{40} = (F_0, 100.MUL_i.balance_i.This_r , H_0)$ $CMV_{42} = (F_0, DIV_i.balance_i.This_r , H_0)$, **$DIV_i = MUL_i / 100$** $CMV_{43} = (F_0, SUB_i.This_r , H_0)$, **$SUB_i = balance_i - DIV$** $CMV_{44} = (F_0, \mathcal{E}, H_1)$, **$This_r \neq null$**, **$Putfield(H_0,H_1,13, This_r, SUB_i)$** $CMV_{47} = (F_0, \mathcal{E}, H_1)$ $CMV_{66} = (F_0, \mathcal{E}, H_1)$

The constraint memory model contributes to automate the test data generation. Indeed, the main goal of the proposed approach [9] is the early detection of infeasible (non-executable) path. However, they do not pay attention to the method called from invalid state. We believe that without taking into account the information contained in the user specification, nothing can guarantee that a given method will not be called from an invalid input state, neither that the implementation is correct relatively to the specification constraints.

If we suppose that the pre-state conditions requires that the *amount parameter* and the *balance attribute* must always be positive, and the *amount parameter* of the method *withdraw()* shall not exceed the balance. Indeed, without adding the pre-state constraints implied by the specification, the labelling of the constraints system will only give the values to the parameters without considering an invocation context that matches the contract associated to the method under test. For example, if the labelling process instantiates the *balance attribute to 100 and* the *amount parameter to -50 for our execution path example,* this execution path is not respecting the pre-state conditions. In this sense, we propose to exploit also the user specifications expressed in the form of precondition, postcondition and class invariant.

4. Testing Method

Combining specification-based testing and white box testing make it possible to test the behavior of the application and also the internal working of the SUT. However, the source code is not always available even more for commercial software. In this context, we propose to exploit firstly, the information contained in the Bytecode of the application to which we have always access and to exploit secondly, the information contained in the user specification. In one hand, the user specification allows us to detect if there are any inconsistencies between the Bytecode program and its specification. On the other hand, having the Bytecode of the program structure will allow us to apply basis path coverage criteria on the execution Graph and to extract its paths which may contain the program-specification inconsistencies, if there are any.

In order to verify the application program from its Java Bytecode and its user specification, the application and its specification must be expressed in the same level of abstraction. To deal with this problem, we propose to encode the specification in the class file using the Static Bytecode instrumentation, i.e. Static Bytecode instrumentation [20] inserts all instrumentation code before the program we want to instrument starts execution.

Our proposal presents two principals steps: firstly, we put the method in the invocation context and then we must therefore make sure that the test process generates valid data. In the second step, we use the post-condition as test oracle for validate the output result. Our aim is to detect if there is a non-conformance between a Method Under Test (MUT) and its specification.

As it is shown in Fig. 4, our Testing Architecture can be divided in Four processing components:

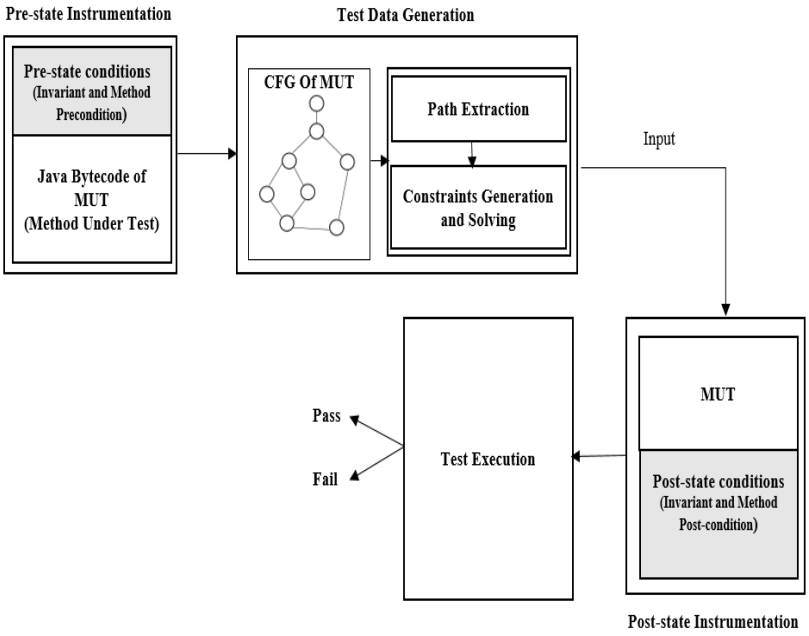

Fig. 4. Testing architecture.

4.1. Instrumented Pre-state

In order to inject the specification in the method that we want to test, we add the precondition and the invariant in form of Bytecode in the beginning of the instructions that perform the operation of the current testing method. As a result, we obtain a method augmented with its pre-state constraints at the Bytecode level.

The instrumentation process of the given testing JAVA method with its pre-state specifications, puts the invoked method into valid state using the method precondition and the class invariant.

4.2. Data Testing Generation

We believe that if we want to perform the tests in the specification level, it is more objective to focus on how the input testing data can be used to explore all the states of the target application. In this sense, the CFG (Control Flow Graph) can be considered as fundamental of our testing approach. It brings a global overview of the execution paths that the input data can take during the execution process.

In this context, this module takes as input the Bytecode of testing method instrumented with its pre-state condition (in textual form; i.e. decompiled with *javap tool*). The outputs are the generated tests data from the paths that are conform to their specifications. The data testing generation, shall be carried out in two steps:

Firstly, we represent the specified Java method of any testing class with its CFG that consists of set of its execution paths; and the basis path code coverage technique [21] based of Depth First Search algorithm in the Control Flow Graph (CFG) allow us to extract the execution paths of instrumented testing method. Secondly, the constraint based testing technique [9] is applied to the extracted execution paths to generate

inputs. The constraint generation and solving process of this technique aim at refining input constraint memory for the method under test by finding values for each variable. In our approach, these values must belong to the precondition domain.

As seen in section III, a constraint system is generated from the semantic of method path Bytecodes. Indeed, each java Bytecode instruction is expressed as a relation between two constraints memory variables (CMVs). A path is considered as valid relatively to its specification pre-state if there exists an input I to the program which is in the domain of the method pre-condition and that covers this execution path. In this sense, when the followed path defined by the precondition and the class invariant contains inconsistent constraints, this means that there is no input from the pre-state domain to cover this path and therefore the latter is discarded. Otherwise, a labelling step on the valid input CMV can starts by labeling the input formal parameters of the method under test. It can either be of type reference or primitive type. When all the parameters are fixed, the values of the object attributes have to be labeled to build a complete CMV.

4.3. Instrumented Post-state

Before the test execution, we inject the post-state conditions (in form of Bytecodes) at the end of the method. This module takes as input a Bytecode of testing method and produce a method specified with its post-state conditions. These post-state assertions are used as decision procedure to detect the particular paths that do not respect user specification or imply contradictory assertions.

4.4. Test Execution

The main purpose of running the method under test with the generated test data is to produce the test verdict. In this step, we have firstly injected the post-condition and the post-state invariant in the end of the MUT. After that, we execute the Method Under Test with the generated valid input test data, and we analyze the returned result: if it leads to a postcondition assertion violation error or an invariant assertion violation error; this means that the method doesn't satisfy its specification. Therefore, this method is not being conform to its specification, particularly, the path traversed by this test data may contain this non-conformance (inconsistency).

5. Testing Example

Consider the Bytecode program shown in the Fig. 3 that corresponds to the method withdraw(). This example is selected to illustrate how we instrument the Method Under Test (MUT), with method specification using the Static Bytecode Instrumentation, as well as how the instrumented Method is translated to the Constraint Memory Model. Our test objective is the detection of non-conformance in the execution paths. We suppose that the pre-state conditions require that the balance attribute and the amount parameter of the method withdraw() must always be positive, and the amount parameter shall not exceed the balance. As post-state, we suppose that the remaining balance is the result of the amount withdrawn from the balance that existed before the transaction.

In order to inject the pre/post specification in the class file, we use the ASM Library [22] for manipulating Bytecodes. In particular, we benefit from the methods visitCode() and visitMaxs() to detect the beginning and the end of method's Bytecode. Our Pre-state Instrumentation Module overrides the method visitCode() of the super-class MethodVisitor. In this context, we can add the pre-state conditions (i.e the precondition and the invariant) in the beginning of the method. Whereas the method visitInsn() is overridden in the Post-state instrumentation module to add the Bytecode instructions (opcodes) corresponding to the post-state conditions (i.e. the post-condition and the invariant).

The Fig. 5 shows the new form of the withdraw() method instrumented with the pre-state constraints. The class invariant, and the precondition are inserted before the withdraw() method body. Now, where the Pre-conditions has been inserted, a basis set of method execution paths will be translated to the Memory Model. The goal is to guide the generation of test data, i.e. generate only valid inputs.

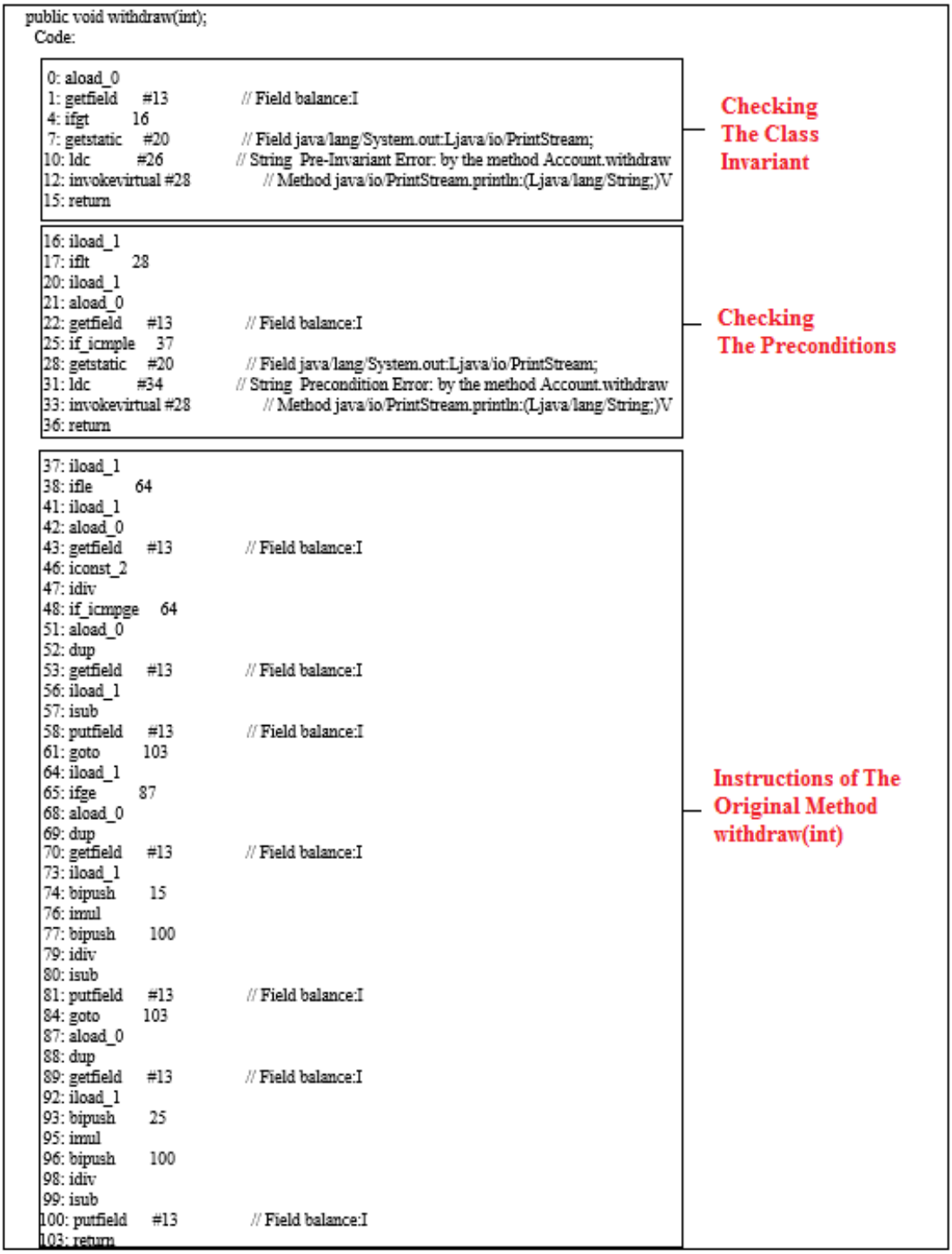

Fig. 5. Method withdraw specified with pre-state specifications.

The Fig. 6 illustrates the control flow graph of method withdraw instrumented with pre-state constraints, i.e. the invariant and the precondition, generated from the class file.

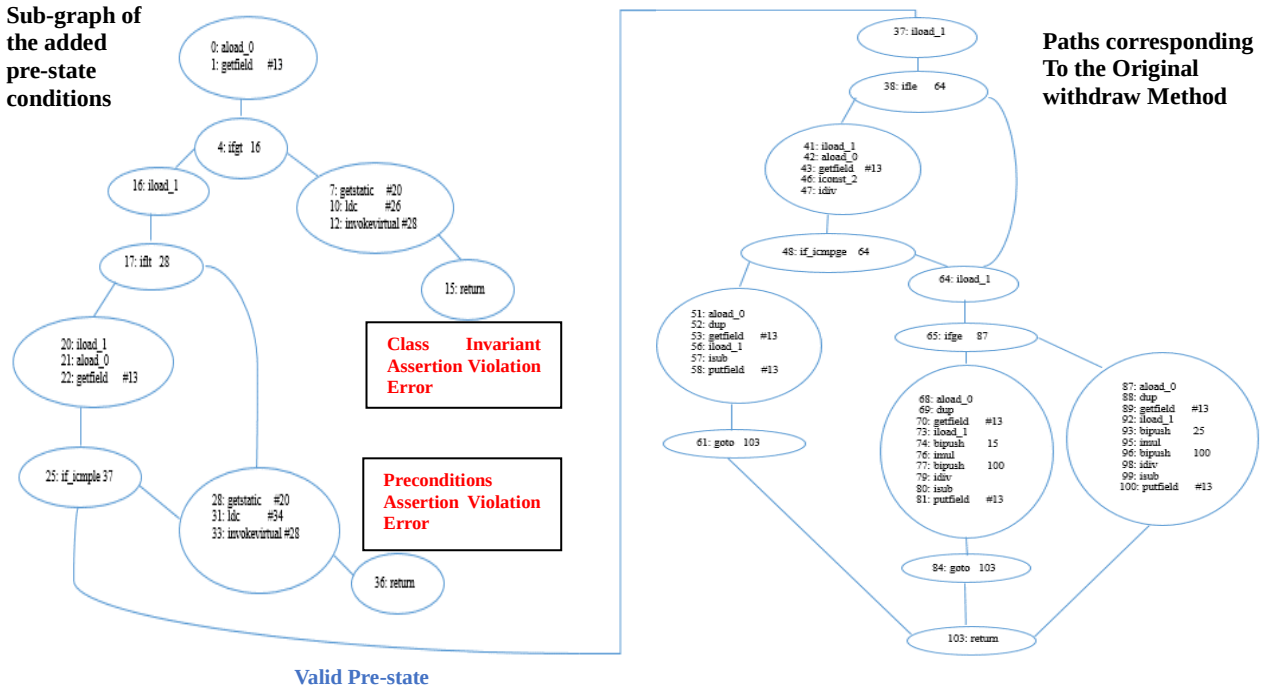

Fig. 6. The control flow graph of method withdraw specified with its pre-state.

After that the control flow graph is generated, the basis following set of paths are extracted (independent paths of the Control Flow Graph):

Path 1: 0 – 1 – 4 – 7 – 10 – 12 – 15
Path 2: 0 – 1 – 4 – 16 – 17 – 28 – 31 – 33 – 36
Path 3: 0 – 1 – 4 – 16 – 17 – 20 – 21 – 22 – 25 – 28 – 31 – 33 – 36.

Path 4: 0 – 1 – 4 – 16 – 17 – 20 – 21 – 22 – 25 – 37 – 38 – 41 – 42 – 43 – 46 – 47 – 48 – 51 – 52 – 53 – 56 – 57 – 58 – 61 – 103.

Path 5: 0 – 1 – 4 – 16 – 17 – 20 – 21 – 22 – 25 – 37 – 38 – 41 – 42 – 43 – 46 – 47 – 48 – 64 – 65 – 68 - 69 – 70 – 73 – 74 – 76 – 77 – 79 – 80 – 81 – 84 – 103.

Path 6: 0 – 1 – 4 – 16 – 17 – 20 – 21 – 22 – 25 – 37 – 38 – 41 – 42 – 43 – 46 – 47 – 48 – 64 – 65 – 87 – 88 – 89 – 92 – 93 – 95 – 96 – 98 – 99 – 100 – 103

Path 7: 0 – 1 – 4 – 16 – 17 – 20 – 21 – 22 – 25 – 37 – 38 – 64 – 65 – 68 - 69 – 70 – 73 – 74 – 76 – 77 – 79 – 80 – 81 – 84 – 103.

Path 8: 0 – 1 – 4 – 16 – 17 – 20 – 21 – 22 – 25 – 37 – 38 – 64 – 65 – 87 – 88 – 89 – 92 – 93 – 95 – 96 – 98 – 99 – 100 – 103

Step 1: Constraint Generation and Valid Input Generation

In order to generate input data from the paths that are valid relatively to the pre-state, we present firstly the generated constraints from the method execution paths augmented with their valid pre-state, i.e. valid precondition and valid invariant. And Then, the path and pre-state constraints consistency is checked; if they are consistent a valid input data is generated.

As seen in Fig. 6, we note that the paths: Path1, Path2 and Path3, that respectively represent the invariant and precondition assertion violation error (i.e. those paths raise a message error when the preconditions or

the invariant are violated), are not taken into consideration, and therefore can be eliminated in this step. The instructions of the paths that begin with valid pre-state (i.e. Path4, Path5, Path6, Path7 and Path8) are translated to their corresponding constraint models.

The initial state: $M_{init} = (F_0, \mathcal{E}, H_0)$, $F_0 = \{0 \rightarrow This_r, 1 \rightarrow amount_i\}$,

- **Constraint generated from the Path 4 augmented with valid Pre-state**

Memory Constraints of the instructions representing the constraint imposed by the Invariant

$CMV_0 = (F_0, This_r, H_0)$
$CMV_1 = (F_0, balance_i, H_0)$
$CMV_4 = (F_0, \mathcal{E}, H_0)$, $\boldsymbol{balance_i > 0}$

Memory Constraints of the instructions representing the constraint imposed by the Pre-condition

$CMV_{16} = (F_0, amount_i, H_0)$
$CMV_{17} = (F_0, \mathcal{E}, H_0)$, $\boldsymbol{amount_i > 0}$
$CMV_{20} = (F_0, amount_i, H_0)$
$CMV_{21} = (F_0, This_r.amount_i, H_0)$
$CMV_{22} = (F_0, balance_i.amount_i, H_0)$
$CMV_{25} = (F_0, \mathcal{E}, H_0)$, $\boldsymbol{amount_i < balance_i}$

Memory Constraints of the instructions representing Path 4 from the original method 'withdraw'

$CMV_{37} = (F_0, amount_i, H_0)$
$CMV_{38} = (F_0, , \mathcal{E}, H_0)$, $\mathbf{amount_i > 0}$
$CMV_{41} = (F_0, amount_i, H_0)$,
$CMV_{42} = (F_0, This_r. amount_i \quad H_0)$,
$CMV_{43} = (F_0, balance_i.amount_i, H_0)$,
$CMV_{46} = (F_0, 2.balance_i.amount_i, H_0)$,
$CMV_{47} = (F_0, DIV_i.amount_i, \quad H_0)$, $DIV_i = balance_i/2$
$CMV_{48} = (F_0, \mathcal{E}, H_0)$, $\mathbf{amount_i < DIV_i}$
$CMV_{51} = (F_0, This_r, H_0)$,
$CMV_{52} = (F_0, This_r.This_r, H_0)$,
$CMV_{53} = (F_0, balance_i.This_r, H_0)$, $\mathbf{This_r \neq null, (This_r, balance_i) \in H}$,
$CMV_{56} = (F_0, amount_i.balance_i.This_r, H_0)$
$CMV_{57} = (F_0, SUB_i.This_r, H_0)$, $\boldsymbol{SUB_i = balance_i - amount_i}$
$CMV_{58} = (F_0, \mathcal{E}, H_1)$, $\mathbf{This_r \neq null}$, $\mathbf{Putfield(H_0, H_1, 13, This_r, SUB_i)}$
$CMV_{61} = (F_0, \mathcal{E}, H_1)$,
$CMV_{103} = (F_0, \mathcal{E}, H_1)$,

- **Constraint generated from the Path 5 augmented with valid Pre-state**

Memory Constraints of the instructions representing the constraint imposed by the Invariant

$CMV_0 = (F_0, This_r, H_0)$
$CMV_1 = (F_0, balance_i, H_0)$
$CMV_4 = (F_0, \mathcal{E}, H_0)$, $\boldsymbol{balance_i > 0}$

Memory Constraints of the instructions representing the constraint imposed by the Pre-condition

$CMV_{16} = (F_0, amount_i, H_0)$
$CMV_{17} = (F_0, \mathcal{E}, H_0)$, $\boldsymbol{amount_i > 0}$
$CMV_{20} = (F_0, amount_i, H_0)$
$CMV_{21} = (F_0, This_r.amount_i, H_0)$
$CMV_{22} = (F_0, balance_i.amount_i, H_0)$
$CMV_{25} = (F_0, \mathcal{E}, H_0)$, $\boldsymbol{amount_i < balance_i}$

Memory Constraints of the instructions representing Path 5 from the original method 'withdraw'

$CMV_{37} = (F_0, amount_i, H_0)$
$CMV_{38} = (F_0, , \mathcal{E}, H_0)$, $\mathbf{amount_i > 0}$
$CMV_{41} = (F_0, amount_i, H_0)$,

CMV_{42} = (F_0, $This_r$. $amount_i$ H_0),
CMV_{43} = (F_0, $balance_i$.$amount_i$, H_0),
CMV_{46} = (F_0, 2.$balance_i$.$amount_i$, H_0),
CMV_{47} = (F_0, DIV_i.$amount_i$, H_0), $DIV_i = balance_i/2$
CMV_{48} = (F_0, $\mathcal{E}$, H_0), **$amount_i$ > DIV_i**
CMV_{64} = (F_0, $amount_i$, H_0)
CMV_{65} = (F_0, $\mathcal{E}$, H_0), **$amount_i$ < 0**
CMV_{68} = (F_0, $This_r$, H_0),
CMV_{69} = (F_0, $This_r$.$This_r$, H_0),
CMV_{70} = (F_0, $balance_i$.$This_r$, H_0), **$This_r$ ≠ null, ($This_r$, $balance_i$) ϵ H,**
CMV_{73} = (F_0, $amount_i$.$balance_i$.$This_r$, H_0)
CMV_{74} = (F_0, 15.$amount_i$.$balance_i$.$This_r$, H_0)
CMV_{76} = (F_0, MUL_i.$balance_i$.$This_r$, H_0), **MUL_i = 15 * $amount_i$**
CMV_{77} = (F_0, 100.MUL_i.$balance_i$.$This_r$, H_0)
CMV_{79} = (F_0, DIV_i.$balance_i$.$This_r$, H_0), **DIV_i = MUL_i / 100**
CMV_{80} = (F_0, SUB_i.$This_r$, H_0), **SUB_i = $balance_i$ - DIV**
CMV_{81} = (F_0, $\mathcal{E}$, H_1), **$This_r$ ≠ null**, **Putfield(H_0,H_1,13, $This_r$, SUB_i)**
CMV_{84} = (F_0, $\mathcal{E}$, H_1)
CMV_{103} = (F_0, $\mathcal{E}$, H_1)

CMV_{51} = (F_0, $This_r$, H_0),
CMV_{52} = (F_0, $This_r$.$This_r$, H_0),
CMV_{53} = (F_0, $balance_i$.$This_r$, H_0), **$This_r$ ≠ null, ($This_r$, $balance_i$) ϵ H,**
CMV_{56} = (F_0, $amount_i$.$balance_i$.$This_r$, H_0)
CMV_{57} = (F_0, SUB_i.$This_r$, H_0), **SUB_i = $balance_i$ – $amount_i$**
CMV_{58} = (F_0, $\mathcal{E}$, H_1), **$This_r$ ≠ null**, **Putfield(H_0,H_1,13, $This_r$, SUB_i)**
CMV_{61} = (F_0, $\mathcal{E}$, H_1),
CMV_{103} = (F_0, $\mathcal{E}$, H_1),

- **Constraint generated from the Path 6 augmented with valid Pre-state**

Memory Constraints of the instructions representing the constraint imposed by the Invariant
CMV_0 = (F_0, $This_r$, H_0)
CMV_1 = (F_0, $balance_i$, H_0)
CMV_4 = (F_0, $\mathcal{E}$, H_0), **$balance_i$ > 0**

Memory Constraints of the instructions representing the constraint imposed by the Pre-condition
CMV_{16} = (F_0, $amount_i$, H_0)
CMV_{17} = (F_0, $\mathcal{E}$, H_0), **$amount_i$ > 0**
CMV_{20} = (F_0, $amount_i$, H_0)
CMV_{21} = (F_0, $This_r$.$amount_i$, H_0)
CMV_{22} = (F_0, $balance_i$.$amount_i$, H_0)
CMV_{25} = (F_0, $\mathcal{E}$, H_0), **$amount_i$< $balance_i$**

Memory Constraints of the instructions representing Path 6 from the original method 'withdraw'
CMV_{37} = (F_0 , $amount_i$, H_0)
CMV_{38} = (F_0, , $\mathcal{E}$, H_0), **$amount_i$ > 0**
CMV_{41} = (F_0 , $amount_i$, H_0),
CMV_{42} = (F_0, $This_r$. $amount_i$ H_0),
CMV_{43} = (F_0, $balance_i$.$amount_i$, H_0),
CMV_{46} = (F_0, 2.$balance_i$.+$amount_i$, H_0),
CMV_{47} = (F_0, DIV_i.$amount_i$, H_0), $DIV_i = balance_i/2$
CMV_{48} = (F_0, $\mathcal{E}$, H_0), **$amount_i$ >DIV_i**
CMV_{64} = (F_0, $amount_i$, H_0)
CMV_{65} = (F_0, $\mathcal{E}$, H_0), **$amount_i$ > 0**
CMV_{87} = (F_0, $This_r$, H_0),
CMV_{88} = (F_0, $This_r$.$This_r$, H_0),
CMV_{89} = (F_0, $balance_i$.$This_r$, H_0), **$This_r$ ≠ null, ($This_r$, $balance_i$) ϵ H,**

CMV_{92} = (F_0, $amount_i.balance_i.This_r$, H_0)
CMV_{93} = (F_0, $25.amount_i.balance_i.This_r$, H_0)
CMV_{95} = (F_0, $MUL_i.balance_i.This_r$, H_0), $\boldsymbol{MUL_i = 25 * amount_i}$
CMV_{96} = (F_0, $100.MUL_i.balance_i.This_r$, H_0)
CMV_{98} = (F_0, $DIV_i.balance_i.This_r$, H_0), $\boldsymbol{DIV_i = MUL_i / 100}$
CMV_{99} = (F_0, $SUB_i.This_r$, H_0), $\boldsymbol{SUB_i = balance_i - DIV}$
CMV_{100} = (F_0, $\mathcal{E}$, H_1), $\mathbf{This_r \neq null}$, $\mathbf{Putfield(H_0,H_1,13, This_r, SUB_i)}$
CMV_{103} = (F_0, $\mathcal{E}$, H_1)

- **Constraint generated from the Path 7 augmented with valid Pre-state (same case for Path 8)**

Memory Constraints of the instructions representing the constraint imposed by the Invariant
CMV_0 = (F_0, $This_r$, H_0)
CMV_1 = (F_0, $balance_i$, H_0)
CMV_4 = (F_0, $\mathcal{E}$, H_0), $\boldsymbol{balance_i > 0}$

Memory Constraints of the instructions representing the constraint imposed by the Pre-condition
CMV_{16} = (F_0, $amount_i$, H_0)
CMV_{17} = (F_0, $\mathcal{E}$, H_0), $\boldsymbol{amount_i > 0}$
CMV_{20} = (F_0, $amount_i$, H_0)
CMV_{21} = (F_0, $This_r.amount_i$, H_0)
CMV_{22} = (F_0, $balance_i.amount_i$, H_0)
CMV_{25} = (F_0, $\mathcal{E}$, H_0), $\boldsymbol{amount_i < balance_i}$

Memory Constraints of the instructions representing Path 7 from the original method 'withdraw'
CMV_{37} = (F_0, $amount_i$, H_0)
CMV_{38} = (F_0, , $\mathcal{E}$, H_0), $\mathbf{amount_i < 0}$
CMV_{64} = (F_0, $amount_i$, H_0)
…….

Path Number	Constraints Generated	Generated Valid Test Data
Path 1	$balance_i < 0$ *(this path represents the Invariant Assertion Violation Error)*	Eliminated
Path 2, Path 3	$amount =< 0 \ \wedge amount_i >= balance_i$ *(these two path represents the Precondition Assertion Violation Error)*	Eliminated
Path 4	$balance_i > 0 \ \wedge \ amount_i > 0 \ \wedge amount_i < balance_i \wedge amount_i < balance_i/2 \ \wedge balance_i = balance_i - amount_i$	$amount_i = 40$ $balance_i = 100$
Path 5	$balance_i > 0 \ \wedge \ amount_i > 0 \ \wedge amount_i < balance_i \wedge amount_i > balance_i/2 \ \wedge \ amount_i <= 0 \ \wedge \ balance_i = balance_i - 15/100 * amount_i$	Discarded path (Path's Conflicting constraints)
Path 6	$balance_i > 0 \ \wedge \ amount_i > 0 \ \wedge amount_i < balance_i \wedge amount_i >= balance_i/2 \ \wedge balance_i = balance_i - 25/100 * amount_i$	$amount_i = 150$ $balance_i = 250$
Path 7, Path 8	$balance_i > 0 \ \wedge \ amount_i > 0 \ \wedge amount_i < balance_i \wedge amount_i <= 0 \ \wedge$ ...	Discarded path (the sub- path constraint is conflicting with Precondition constraints)

Constraints Analysis

In the two paths 4 and 6 presented above, the path constraints are not contradictory with the constraints of the invariant and the preconditions. Therefore, valid test data are generated from these paths where both the method pre-states are respected and the execution path constraints are respected.

The paths that traverse the sub path [37, 38, 64, …] (i.e. path 7 and path 8), will be discarded due to the conflicting arisen by with the method precondition constraints and the sub path constraints. In other word, when the constraints of the execution path are contradictory with the pre-state constraints, the path becomes infeasible and therefore must be discarded in our example the path 5 presents the same inconsistency.

Step 2: Test Execution (Non-conformance Detection)

As we have shown before, the post-state is used as test oracle and before the test execution, we instrument the MUT with the Post-state constraints.

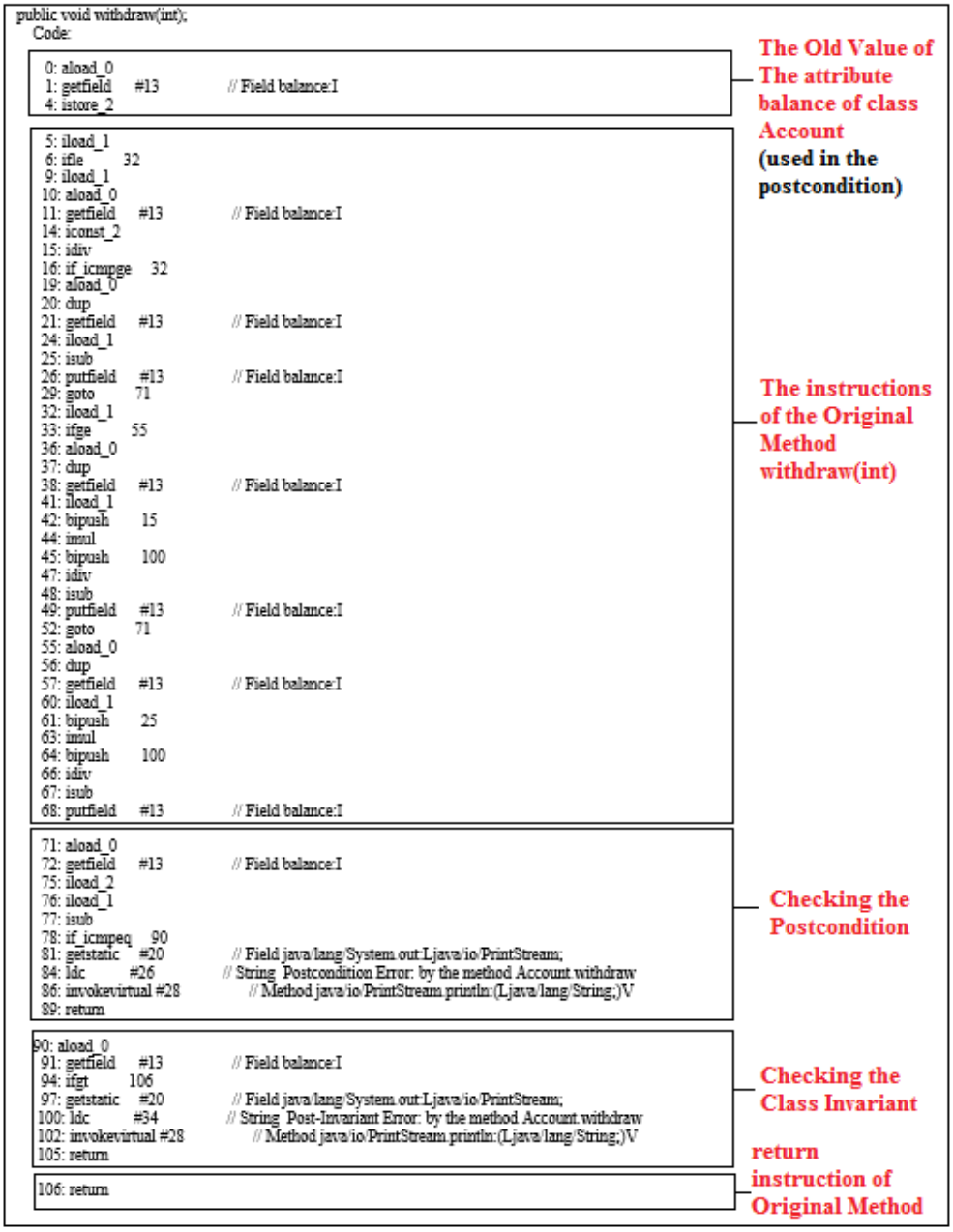

Fig. 7. Method withdraw specified with its post-state specifications.

If we execute the MUT with generated valid input data and the post-condition assertion error is raised, then a non-conformance between the specification and the program is detected. In this case, we can confirm that this non-conformance is detected in the path from which we have generated the input data.

Path Number	**Generated Test Data From the Path**	Test Result
Path 4	$amount_i = 40$ $balance_i = 100$	60 (Post-state Respected)
Path 6	$amount_i = 150$ $balance_i = 250$	PostCondition Assertion Error

The advantage of this method is that we can detect which execution path in the MUT contains the non-conformance. Indeed, in our example we can deduce that a non-conformance is detected in the Path 6; i.e. by executing the program with the generated amount = 150 and *balance = 250*, the result returned by the program *(balance = 213)* does not respect the post-condition *balance = balance – amount.* Therefore, the Method is not conform to its specification and the inconsistency resides in the Path6.

6. Conclusion

In this paper we have present a testing method that consider the constraints of user specification expressed as Invariant Pre/Post Specifications to generate test data and to detect non-conformance of the execution paths in Bytecode level.

We have shown how we express those specifications at the Bytecode level using static Bytecode instrumentation. In one hand, this representation of the specification makes it possible to perform constraint based testing of Java Bytecode method augmented with pre-state conditions; where the preconditions of the method put the method in its invocation context which allows to generate valid structural test data (and consequently to eliminate the paths in which the constraints are contradictory with the pre-state constraints). In other hand, we execute the program instrumented with its post-state condition to look for non-conformance between the method and its specification. The main advantage of this technique is that it shows exactly which execution path that contains this non-conformance. Our work, is now oriented to detect path anomalies for secure testing.

References

[1] Reichl, K., Fischer, T., & Tummeltshammer, P. (2016). Using formal methods for verification and validation in railway. *Proceedings of International Conference on Tests and* Proofs (pp. 3-13).

[2] Posegga, J., & Vogt, H. (1998). Java bytecode verification using model checking. *Proceedings of the OOPSLA'98 Workshop Formal Underpinnings of Java*.

[3] DeMilli, R. A., & Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering, 17(9)*, 900-910.

[4] De Moura, L., & Bjørner, N. (2011). Satisfiability modulo theories: Introduction and applications. *Communications of the ACM, 54(9)*, 69-77.

[5] Vincenzi, A. M. R., Delamaro, M. E., Maldonado, J. C., & Wong, W. E. (2006). Establishing structural testing criteria for java bytecode. *Software: Practice and Experience, 36(14)*, 1513-1541.

[6] Zhao, J. (2000). Dependence analysis of Java bytecode. *Proceedings of the 24th Annual International Conference on Computer Software and Applications, COMPSAC 2000* (pp. 486-491).

[7] Amine, A., Mohammed, B., & Jean-Louis, L. (2014). Generating control flow graph from Java card byte code. *Proceedings of 3rd IEEE International Colloquium in Information Science and Technology (CIST),* (pp. 206-212).

[8] Păsăreanu, C. S., & Rungta, N. (2010). Symbolic PathFinder: Symbolic execution of Java bytecode. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (pp. 179-180).

[9] Charreteur, F., & Gotlieb, A. (2010). Constraint-based test input generation for java bytecode. *Proceedings of IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)* (pp. 131-140).

[10] Charreteur, F., & Gotlieb, A. (2008). Raisonnement à contraintes pour le test de bytecode Java. *JFPC 2008-Quatrièmes Journées Francophones de Programmation par Contraintes* (pp. 11-20).

[11] Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. *Future of Software Engineering*, 85-103.

[12] Benlhachmi, K., & Benattou, M. (2013). A formal model of conformity and security testing of inheritance for object oriented constraint programming. *Journal of Information Security, 4(2),* 113.

[13] Dadeau, F., & Peureux, F. (2011). Grey-box testing and verification of Java/JML. *Proceedings of 2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* (pp. 298-303). IEEE.

[14] Offutt, A. J., Jin, Z., & Pan, J. (1999). The dynamic domain reduction procedure for test data generation. *Software-Practice and Experience, 29(2),* 167-93.

[15] Benattou, M., Bruel, J. M., & Hameurlain, N. (2002). Generating test data from OCL specification. *Proceedings of ECOOP Workshop Integration and Transformation of UML Models*.

[16] Cheon, Y., & Rubio-Medrano, C. E. (2007). Random test data generation for Java classes annotated with JML specifications. *Proceedings of the International Conference on Software Engineering Research and Practice* (pp. 385-392).

[17] Bernard, E., Legeard, B., Luck, X., & Peureux, F. (2004). Generation of test sequences from formal specifications: GSM 11-11 standard case study. *Software: Practice and Experience, 34(10)*, 915-948.

[18] Vincenzi, A. M. R., Wong, W. E., Delamaro, M. E., & Maldonado, J. C. (2003). JaBUTi: A coverage analysis tool for Java programs. *XVII SBES–Simpósio Brasileiro de Engenharia de Software*, 79-84.

[19] Xu, W., Ding, T., & Xu, D. (2014). Rule-based test input generation from Bytecode. *Proceedings of 8th International Conference on Software Security and Reliability* (pp. 108-117). IEEE.

[20] Binder, W., Hulaas, J., & Moret, P. (2007). Advanced Java bytecode instrumentation. *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java* (pp. 135-144). ACM.

[21] Poole, J. (1995). A method to determine a basis set of paths to perform program testing. *US Department of Commerce/National Institute of Standards and Technology*.

[22] Bruneton, E. (2007). *ASM 3.0 A Java bytecode engineering library.* Retrieved from http://download. forge. objectweb. org/asm/asmguide. pdf

Safaa Achour is a Ph.D student of computer science in Faculty of sciences at the University of Ibn Tofail Kénitra, Morocco. she received her master's degree in software quality from the same university in 2012. Her research interests are in the areas of software testing, particularly, in model based testing and Java Bytecode testing.

Mohammed Benattou is professor of computer science in Faculty of sciences at the University of Ibn Tofail Kénitra, Morocco, where he has directed the computer science and telecommunication laboratory. After completing his Ph.D at University of Blaise Pascal Clermont-Ferrand, he has held several positions in her French academic career: University

of Pau, University of Orsay Paris XI, 3IL Institute of IT & Engineering of Limoges and Xlim Laboratory. His research interests are generally in the areas of software testing that include distributed testing, secure testing and model based testing.