International Journal of Software Engineering and Its Applications
Vol.12, No.2 (2018), pp.1-16
http://dx.doi.org/10.21742/ijseia.2018.12.2.01

A Constraint-Based Verification Approach for Java Bytecode Programs

Safaa Achour, Ali Chouenyib and Mohammed Benattou

LASTID Laboratory, Ibn Tofail University, Kénitra, Morocco
safaa.achour@uit.ac.ma, chouenyib@gmail.com, mbenattou@yahoo.fr

Abstract

In this paper, we propose a constraint based analysis technique to detect the inconsistencies between a Java application and its specification at the Bytecode level. The main objective of our approach is not only to exploit the information of the user specification but also the memory constraints generated from the Java Bytecode of the application. Indeed, this allows us to detect the possible non-conformance between a program and its specification and also to explore the execution paths of the application looking at which of them may contain such inconsistencies. Nevertheless, the testing application and the user specification are not in the same level of abstraction. Thereby, we propose to wrap the method under test with its specification expressed as pre/post specifications at the Bytecode level, using the Static Bytecode Instrumentation.

Keywords: *Software verification; Static Bytecode Instrumentation; ASM, Pre/Post Specifications; Java Bytecode; Constraints*

1. Introduction

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 [1]. 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).

Indeed, 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 [2]. Various approaches use constraint solving techniques together with annotated programs either to produce test cases or to verify program correctness [3].

Although the verification process, which are based on mathematical proofs, allow guaranteeing the absence of certain classes of errors, the testing methods can stimulate properly the behavior of software by applying inputs and checking if the specifications are respected in the output. However, tests reliability depends on the count of test oracles and the efficiency of the input data to parse the maximum states of the SUT. In this context, constraint-based testing introduced by Offutt in 1991 [4], combines a symbolic execution and dynamic constraint solving [5] in order to generate test inputs.

However, often 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

Received (September 15, 2017), Review Result (February 6, 2018), Accepted (February 18, 2018)

ISSN: 1738-9984 IJSEIA

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 [6]. 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 programs [7][8], performing symbolic execution of Bytecode [9], or using constraint based techniques to generate test inputs from java Bytecode programs [10, 11]. However, few of them have been interested in testing the behavior 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 is known, the SUT and the user specification are not at the same level. The program is at Bytecode level whereas the specification is in high-level of abstraction. In Java software context, we deal with two issues: how can we specify Java Bytecode programs, and how we can check execution paths of the target application to detect a non-conformance with the user specification.

In this paper we propose, to specify the Java Bytecode program at the Bytecode level using Static Bytecode Instrumentation. The main idea of the proposed work is to wrap the Method Under Test with its user specification by adding (statically) assertions in the form of precondition, post-condition and invariant and to check the expected behavior of the given method at the Bytecode level. Our approach aims to detect non-conformance between a given Byte code program and its specification in the context of unit testing.

This paper is organized as follow: Section 2 presents the Java Bytecode testing problem and its related work, Section 3 gives a brief description of the Java Virtual Machine (JVM) representation and the existing memory constraint model, Section 4 describes our proposed testing approach, Section 5 presents an example of Byte code program, its specification, and describe how the Wrapped Method of the given example is translated to the Constraint Memory Model, finally section 6 gives some concluding remarks and outline our future works.

2. Related Work

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 [8], 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 [12], 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 [9], 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 [13] a new rule-based testing (RBT) approach to automated generation of test inputs from Java Bytecode without using fitness functions.

In [10], the authors describe a goal-oriented method that aims at building an input state of the Java Virtual Machine that can drive program execution towards a given location within the Bytecode. 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.

There are several approaches to automatic test data generation based on formal specifications. In [14], 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 [15], propose an automated random testing method as a practical tool to assure the correctness of interface specifications. In [16], 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.

In this context, we propose to specify the Java Bytecode program at the Bytecode level using the Static Bytecode Instrumentation. We explore the notion of constrained memory variable [10, 11] to generate constraint system from the method wrapped with pre/post specifications. Indeed, contrary to [10] where the authors exploit the constraint memory model to early detect infeasible paths in the Bytecode program, our approach aims to detect non-conformance between a given Bytecode program and its specification in the context of unit testing, such inconsistency is detected by checking the satisfiability of the path constraints augmented with its pre-state and the negation of its post-state.

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 [10,11] uses Constrained Memory Variables (CMV) to represent JVM states.

The JVM states represent runtime data storage locations such as registers, the operand stack and the 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 Figure 1 shows an example of Java Bytecode method execution.

The modelling by constraints of Java Bytecode has required the definition of memory model [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. A CMV contains data storage locations where data can be represented by variable along the domain. As it is represented formally in [10] 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.

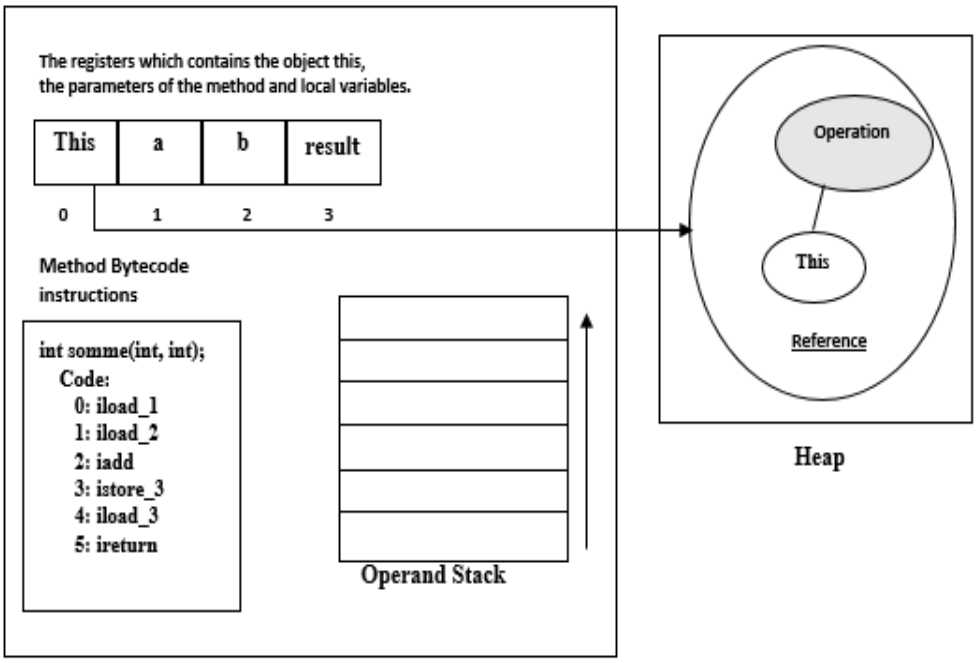

Figure 1. Example of Java Bytecode Method Execution in the JVM

Each java Bytecode instruction of the program is seen as a 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 M_j before the activation of Bytecode and the CMV M_k 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. Integers 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 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 a pair element; 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.

int Add(int, int); **Code:** **0: iload_1** **1: iload_2** **2: iadd** **3: istore_3** **4: iload_3** **5: ireturn** **}**	$F_0 = \{0 \rightarrow Thisr, 1 \rightarrow a, 2 \rightarrow b, 3 \rightarrow result\}$ $CMV_{init} = (F_0, \mathcal{E}, H_0)$, $CMV_0 = (F_0, a_r, H_0)$, $CMV_1 = (F_0, b.a_r, H_0)$, $CMV_2 = (F_0, ADD_i, H_0)$, $ADD_i = a + b$ $CMV_3 = (F_1, \mathcal{E}, H_0)$, $result = ADD_i$ $CMV_4 = (F_1, result_r, H_0)$, $CMV_5 = (F_1, \mathcal{E}, H_0)$,

Figure 2. Java Bytecode Example and its Memory Constraint Model

Figure 2 shows an example of the java Byte code of the method Add of class Operation, and gives its correspondent Constraint Memory Model.

The constraint memory model contributes to automate the test data generation. Indeed, the main purpose of the proposed approach [10] is to deal with the reachability problem, *i.e.*, the early detection of infeasible (non-executable) path. However, they do not pay attention to the method called from an invalid state. We believe that without taking into account the information contained in the user specification, nothing can help to detect the inconsistencies between a given method and its specification, if there are any. In this sense, we propose to exploit also the user specifications to verify the expected behavior of the target application.

4. Verification Approach

Combining specification-based techniques and white box methods make it possible to verify 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 sense, 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. In the other hand, having the Bytecode program structure will help us to know the paths that may contain these inconsistencies.

In order to verify the application program from its Java Bytecode and its user specification, both the testing program and its specification must be expressed in the same level of abstraction. Our proposal is first to specify the application at the Bytecode level, and then to perform the verification of the application. In this context, we propose to inject the specification in the class file using Static Bytecode Instrumentation. The idea is to wrap the Method Under Test with user specification in order to specify the behavior of java classes (java methods) by adding assertions in the form of precondition, post-condition and invariant.

In the second step, the instructions of the execution paths of the wrapped method are translated to their correspondent constraint model. Finally, we check from this model if there is a non-conformance presented in the execution paths relatively to the pre/post specifications. The negation of the post-condition is used to detect the non-conformance.

4.1. Injection of the Specification

Specification languages such JML has being used for documenting and assuring the correctness of the program [17]. The assertions of these specification languages are written as Boolean expressions of the underlying programming language, and can be executed and thus checked at runtime.

However, such specification languages are often restricted to the code source of the application. We believe that it is necessary to have a way to describe the functional behavior of a given application at the Bytecode level for several reasons:

- Most of the time, the class file is delivered to the client or the tester without the specified source.
- Some applications are developed directly at the Bytecode Level
- The code receiver checks the executable code than its source code.
- The Byte code proofs aim to guaranty that some security requirement achieve correctly the protection from malicious code

In this context, we propose to formally express the specification as Pre/Post conditions and class invariant at the Bytecode Level using Static Bytecode Instrumentation. In our approach, the specification is concerning one target Java application (method) at the Bytecode level. Every method of the class file contains a sequence of Bytecode instructions. As illustrated in Figure 3, our idea is to wrap the method that we want to verify with its specification:

- ✓ Precondition and invariant (in form of Bytecodes) are injected, statically, before the instructions performing the operation of the original method.
- ✓ Post-condition and post-invariant constraints are inserted at the end of the method just before the execution of return instruction of the original method.

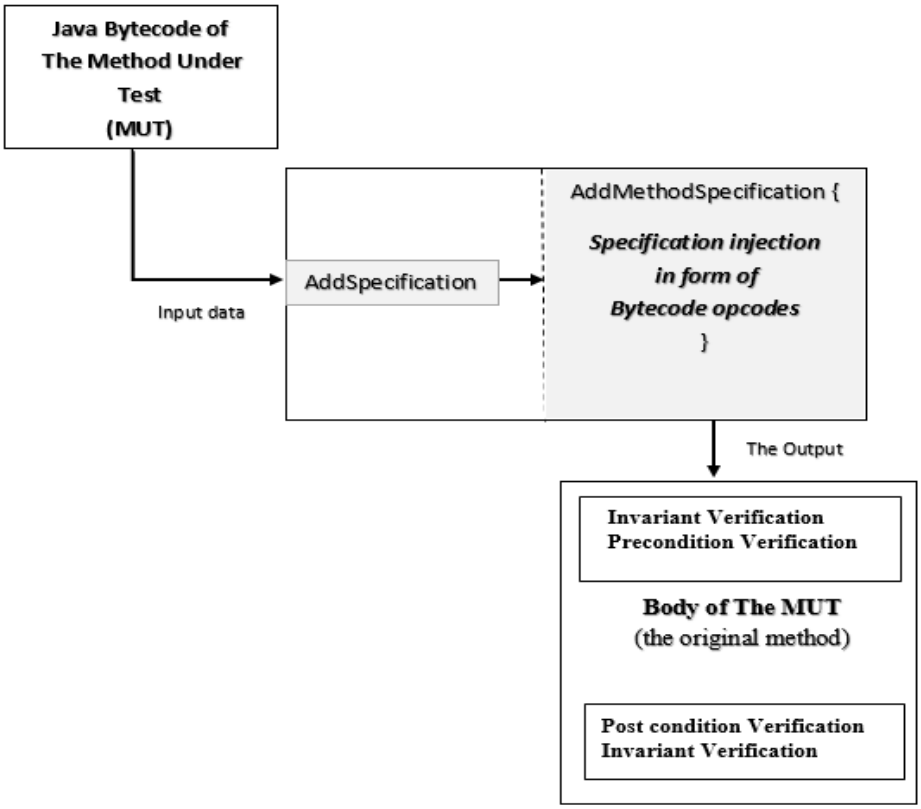

Figure 3. Injection of the User Specification

As it is shown in Figure 3, we have implemented the *AddSpecification* Module that reads and modifies the class file. The calling of the *AddMethodSpecification* searches for the method of interest, and then inserts Bytecode instructions corresponding to the pre/post conditions in form of opcodes. In our case, the Bytecode instrumentation does not need the program source code. In order to implement our approach we deploy ASM library [18] to manipulate the Java application class files using Static Bytecode Instrumentation, and in particular, the methods *visitCode()* and *visitMaxs()* are used to detect the beginning and the end of method's Bytecode. The method *visitCode()* of the super-class *MethodVisitor* is overridden so that 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 order to add the Bytecode instructions (opcodes) corresponding to the post-state conditions, *i.e.*, the post-condition and the invariant. Note that the post-condition and invariant are added at the end just before the return instruction.

4.2. Verification Process

We believe that if we want to perform the verification 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 verification approach. It brings a global overview of the execution paths that the input data can take during the execution process.

The instrumentation process of the given testing JAVA method with its specifications, puts the invoked method into valid state using the method precondition and the class invariant, and makes it possible to check if the method ends in the state expected by the post-state.

Firstly, we represent the wrapped Java method of any testing class with its CFG; and the basis path testing technique [19] based of Depth First Search algorithm in the Control Flow Graph (CFG) allows us to extract the execution paths of the specified testing method.

Secondly, a constraint system is generated from the Bytecodes semantic of the extracted paths of the MUT augmented with the method Pre-state (*i.e.*, the precondition and the invariant), and the negation of the method post-state (*i.e.*, the non(postcondition)):

$$(\text{precondition} \wedge \text{invariant}) \wedge \text{original method path constraint} \wedge (\neg \text{postcondition})$$

Indeed, as seen in section III, each java Bytecode instruction is expressed as a relation between two constraint memory variables (CMVs).

Finally, we check the satisfiability of the given generated constraint system. If it does not include contradictory constraints, this means that: (1) The pre-state constraints are respected, (2) The negation of the post-state is satisfied and then the path verification results in post-state violation error. Consequently, we can deduce that the execution path of the original method is not conform to its specification. Indeed, an invalid path is an execution path that begins in valid state and ends in state where it does not respect the post-condition. We can confirm that, if an invalid path is detected, the invoking method does not conform to its specification. The main advantage of this approach is that it shows exactly which execution path of the method under test does not respect the user specification, at the Bytecode level. We mention that the constraints consistency is checked on the fly in the same way as in [10].

5. Verification Example

Consider the Java program of Figure 4 that implements the class Account. The Bytecode program shown in the Figure 5 correspond to the method withdraw(int). This example is selected to illustrate how we wrap the Method Under Test (MUT), withdraw(int), with pre/post specification using the Static Bytecode Instrumentation, as well as how the Wrapped Method is translated to the Constraint Memory Model. Our test objective is the detection of non-conformance between the MUT and its specification.

```
public class Account {
  private int balance;

public Account(int balance){
     this.balance = balance;
}

public void withdraw(int amount){
   if(amount >= 100)
           balance = balance - amount;
   else
     balance = balance - amount * 25/100;
    }}
                                  //.....
}
```

Figure 4. Example in Java Source Code

We suppose that the pre-state conditions require that the amount must be positive and the amount withdrawn 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. The balance attribute must always be positive.

```
public void withdraw(int);
   Code:
     0: iload_1
     1: bipush       100
     3: if_icmple    19
     6: aload_0
     7: dup
     8: getfield     #13              // Field balance:I
    11: iload_1
    12: isub
    13: putfield     #13              // Field balance:I
    16: goto         35
    19: aload_0
    20: dup
    21: getfield     #13              // Field balance:I
    24: iload_1
    25: bipush       25
    27: imul
    28: bipush       100
    30: idiv
    31: isub
    32: putfield     #13              // Field balance:I
    35: return
```

Figure 5. Example in Bytecode (of the withdraw original method)

The Figure 6 shows the new form of wrapped *withdraw(int)* method. The class invariant which assumes that the balance is always positive, and the precondition that requires that the amount should be positive and the withdrawn amount must not exceed the existent balance are inserted before the *withdraw(int)* method body.

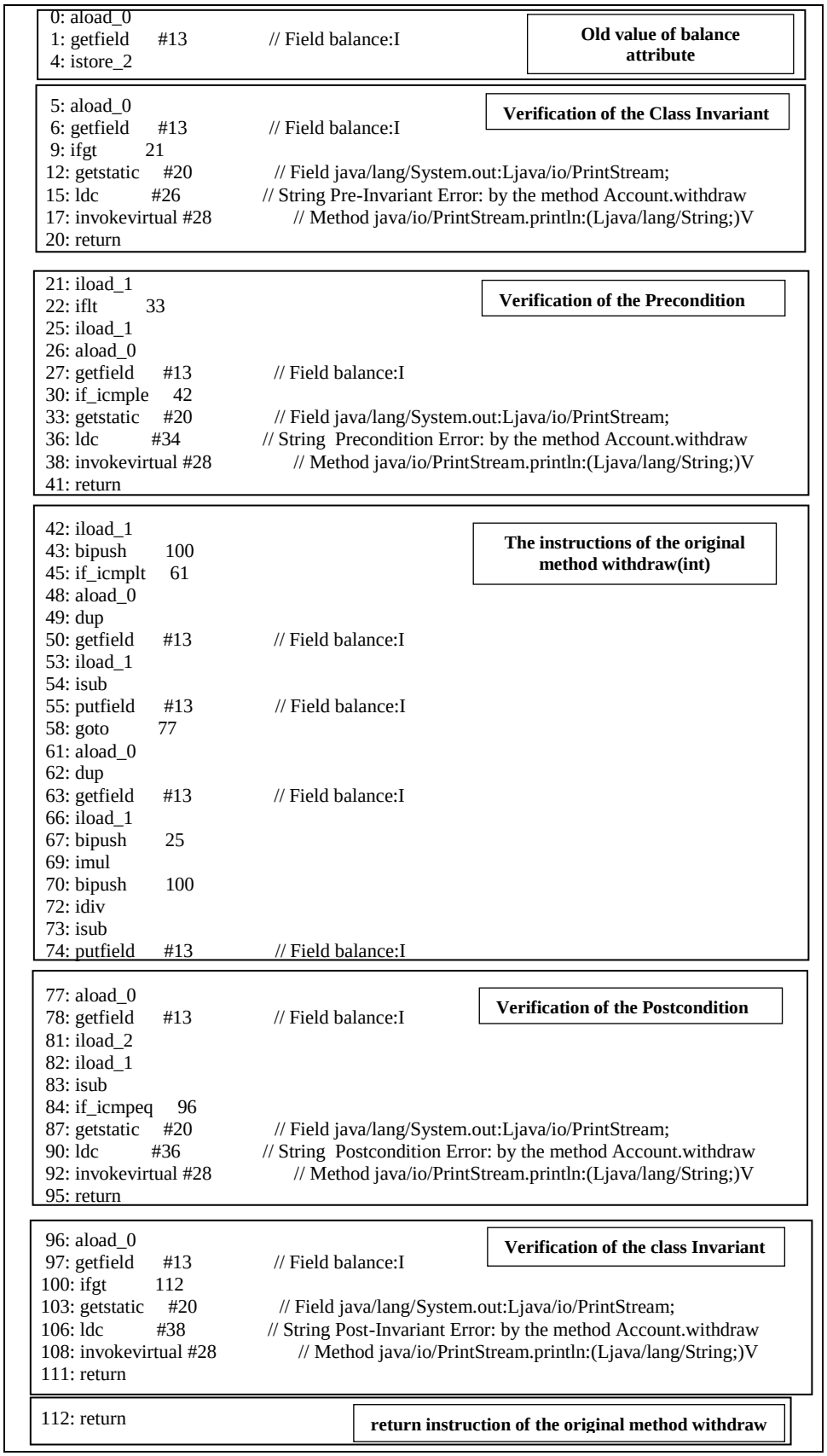

Figure 6. Method Withdraw Instrumented with Pre/Post Specifications

Whereas the post-condition and the invariant constraints are inserted at the end, before the return instruction of the method under test. We mention that the attribute balance ***old value*** is injected in the beginning of the method to save the pre-state value of the attribute balance. The ***old value*** is usually used in the post-condition of the method.

We note that the paths that raise an invariant assertion error or a precondition assertion error, as seen in the Figure 7, are not taken into consideration, and therefore are discarded.

In the following, we give the example of constraint memory system of the execution paths of the wrapped method. If any of them contains an inconsistency with its specification, a non-conformance of the method withdraw is then detected.

The withdraw method initial state is: $M_{init} = (F_{0,} \mathcal{E}, H_0)$, $F_0 = \{This_r, amount_i, old\$balance_i\}$; As said before, $\boldsymbol{old\$balance_i}$ refers to the value of balance in its pre-state.

Consider the second path [0 – 1 – 4 – 5 – 6 – 9 – 21 – 22 – 25 – 26 – 27 – 30 – 42 – 43 – 45 – 48 – 49 – 50 – 53 – 54 – 55 – 58 – 77 – 78 – 81 – 82 – 83 – 84 – 87 – 90 – 92 – 95] of the wrapped method *withdraw(int)*.

$CMV_0 = (F_{0,} This_r , H_0)$

$CMV_1 = (F_{0,} balance_i , H_0)$

$CMV_4 = (F_1 , \mathcal{E} , H_0)$, $\boldsymbol{old\$balance_i = balance_i}$;

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

$CMV_5 = (F_{1,} This_r , H_0)$

$CMV_6 = (F_{1,} balance_i , H_0)$

$CMV_9 = (F_1, \mathcal{E}, H_0)$, $\boldsymbol{balance_i > 0}$

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

$CMV_{21} = (F_1, amount_i , H_0)$

$CMV_{22} = (F_1, \mathcal{E}, H_0)$, $\boldsymbol{amount_i \geq 0}$

$CMV_{25} = (F_1, amount_i , H_0)$

$CMV_{26} = (F_1, This_r.amount_i , H_0)$

$CMV_{27} = (F_1, balance_i.amount_i , H_0)$

$CMV_{30} = (F_1, \mathcal{E}, H_0)$, $\boldsymbol{amount_i \leq balance_i}$

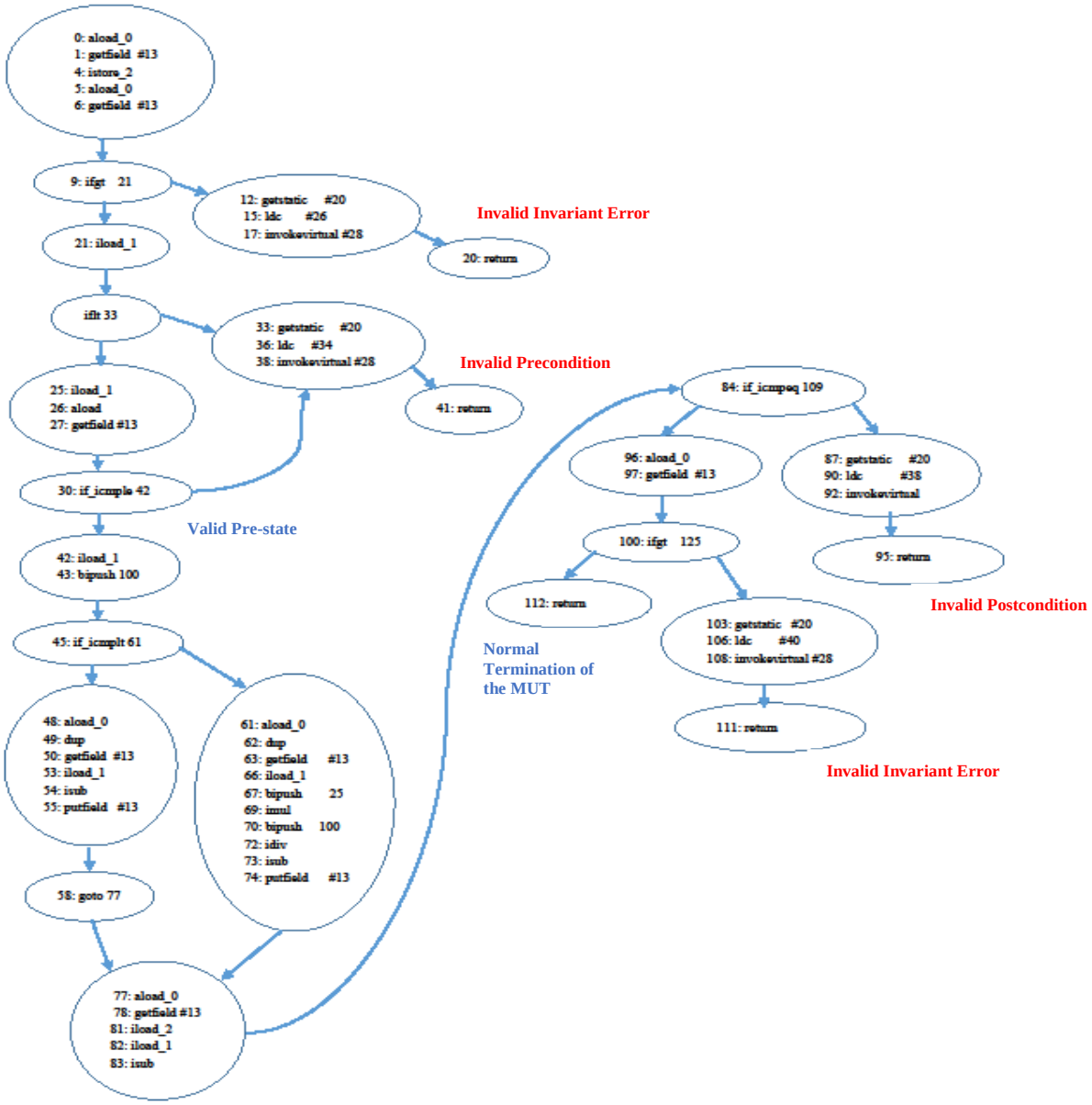

Figure 7. The Control Flow Graph of Method Withdraw Wrapped with its Pre, Post-States

Memory Constraints of the instructions representing the second path of the original method 'withdraw'

$CMV_{42} = (F_1 , amount_i , H_0)$

$CMV_{43} = (F_1, 100.amount_i, H_0)$,

$CMV_{45} = (F_1, \mathcal{E}, H_0)$, $\boldsymbol{amount_i \geq 100}$

$CMV_{48} = (F_1, This_r , H_0)$,

$CMV_{49} = (F_1, This_r.This_r , H_0)$,

CMV_{50}= (F_1, $balance_i.This_r$, H_0), **$This_r \neq$ null, ($This_r$, $balance_i$) ϵ H_0,**

CMV_{53}= (F_1, amounti.balancei.Thisr , H_0)

CMV_{54}= (F_1, SUBi.Thisr , H_0), **SUB_i=$balance_i$-$amount_i$**

CMV_{55} = (F_1, $\mathcal{E}$, H_1), **Thisr ≠ null, Putfield(H0,H1,13, $This_r$, SUB_i)**

CMV_{58} = (F_1, $\mathcal{E}$, H_1),

Memory Constraints of the instructions representing the constraint of the non Post-condition

CMV_{77} = (F_1, $This_r$, H_1),

CMV_{78}= (F_1, $balance_i$, H_1), **$This_r \neq$ null, ($This_r$, $balance_i$) ϵ H_1,**

CMV_{81} = *(F_1, $old\$balance_i.balance_i$, H_1),*

CMV_{82} = *(F_1, $amount_i.old\$balance_i.balance_i$, H_1),*

CMV_{83} = *(F_1, $SUB_i.balance_i$, H_1),* ***SUB_i = $old\$balance_i$ - $amount_i$***

CMV_{84} = (F_1, $\mathcal{E}$, H_1), ***balance != SUB_i***

CMV_{87} = (F_1, java/lang/System.out,H_1)

CMV_{90} = (F_1,"Postcondition Error..". java/lang/System.out,H_1)

CMV_{92} = (F_1, $\mathcal{E}$, H_1),

CMV_{95} = (F_1, $\mathcal{E}$, H_1),

Note that the constraint **Putfield(H_0,H_1,13, $This_r$, SUB_i)** indicates that the attribute $balance_i$ of the current object Account receives the value of the variable SUBi, which changes the state H_0 of the heap to a state H_1. This means that $balance_i = SUB_i$, or more specifically **$balance_i$ = $balance_i$ – $amount_i$.**

The constraint system generated from this execution path is the following:
$old\$balance_i$ = $balance_i$ ∧ $balance_i$ > 0 ∧ $amount_i$ ≥ 0 ∧ $amount_i$ ≤ $balance_i$ ∧ $amount_i$ ≥ 100 ∧ $balance_i$=$balance_i$-$amount_i$ ∧ $balance_i$!= $old\$balance_i$ - $amount_i$

As this constraint system is unsatisfiabe, the corresponding execution path does not contain any inconsistency. In fact, the domain of valid input data that traverse this execution path is restricted to the values that respect both the constraints of the method pre-state and the constraints of the given path, i.e. ***$amount_i$ < $balance_i$ ∧ $amount_i$ ≥ 100 ∧ $balance_i$ > 0***; and for these values, the latter terminates in final state that is conflicting with the negation of the postcondition. Indeed,
$old\$balance_i$ = $balance_i$ ∧ $balance_i$ > 0 ∧ $amount_i$ ≥ 0 ∧ $amount_i$ ≤ $balance_i$ ∧ $amount_i$ ≥ 100 ∧ $balance_i$=$balance_i$-$amount_i$ ⇏ $balance_i$!= $old\$balance_i$ - $amount_i$.

So, we pass to check the following path.

Consider the second path [0 – 1 – 4 – 5 – 6 – 9 – 21 – 22 – 25 – 26 – 27 – 30 – 42 – 43 – 45 –61 – 62 – 63 – 66 – 67 – 69 – 70 – 72 – 73 – 74 – 77 – 78 – 81 – 82 – 83 – 84 – 87 – 90 – 92 – 95] of the wrapped method *withdraw(int)*.

CMV_0 = *(F_0, $This_r$, H_0)*

$CMV_1 = (F_0, balance_i , H_0)$

$CMV_4 = (F_1 , \mathcal{E} , H_0)$, $\mathbf{old\$balance_i = balance_i}$;

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

$CMV_5 = (F_1, This_r , H_0)$

$CMV_6 = (F_1, balance_i , H_0)$

$CMV_9 = (F_1, \mathcal{E}, H_0)$, $\mathbf{balance_i > 0}$

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

$CMV_{21} = (F_1, amount_i , H_0)$

$CMV_{22} = (F_1, \mathcal{E}, H_0)$, $\mathbf{amount_i \geq 0}$

$CMV_{25} = (F_1, amount_i , H_0)$

$CMV_{26} = (F_1, This_r.amount_i , H_0)$

$CMV_{27} = (F_1, balance_i.amount_i , H_0)$

$CMV_{30} = (F_1, \mathcal{E}, H_0)$, $\mathbf{amount_i \leq balance_i}$

Memory Constraints of the instructions representing the second path of the original method 'withdraw'

$CMV_{42} = (F_1 , amount_i , H_0)$

$CMV_{43} = (F_1, 100.amount_i, H_0)$,

$CMV_{45} = (F_1, \mathcal{E}, H_0)$, $\mathbf{amount_i < 100}$

$CMV_{61} = (F_1, This_r , H_0)$,

$CMV_{62} = (F_1, This_r.This_r , H_0)$,

$CMV_{63} = (F_1, balance_i.This_r , H_0)$, $This_r \neq null$, $(This_r , balance_i) \in H_0$,

$CMV_{66} = (F_1, amount_i.balance_i.This_r , H_0)$

$CMV_{67} = (F_1, 25.amount_i.balance_i.This_r , H_0)$

$CMV_{69} = (F_1, MUL_i.balance_i.This_r , H_0)$, $\mathbf{MUL_i = 25 * amount_i}$

$CMV_{70} = (F_1, 100.MUL_i.balance_i.This_r , H_0)$

$CMV_{72} = (F_1, DIV_i.balance_i.This_r , H_0)$, $\mathbf{DIV_i = MUL_i / 100}$

$CMV_{73} = (F_1, SUB_i.This_r , H_0)$, $\mathbf{SUB_i = balance_i - DIV_i}$

$CMV_{74} = (F_1, \mathcal{E}, H_1)$, $\mathbf{This_r \neq null}$, $\mathbf{Putfield(H_0,H_1,13, This_r, SUB_i)}$

Memory Constraints of the instructions representing the constraint of the non Post-condition

$CMV_{77} = (F_1, This_r , H_1)$,

$CMV_{78} = (F_1, balance_i , H_1)$, $\mathbf{This_r \neq null, (This_r , balance_i) \in H_1,}$

CMV_{81} = $(F_1, old\$balance_i.balance_i, H_1)$,

CMV_{82} = $(F_1, amount_i.old\$balance_i.balance_i, H_1)$,

CMV_{83} = $(F_1, SUB_i.balance_i, H_1)$, $\boldsymbol{SUB_i = old\$balance_i - amount_i}$

CMV_{84} = $(F_1, \mathcal{E}, H_1)$, $\boldsymbol{balance\ != SUB_i}$

CMV_{87} = $(F_1, java/lang/System.out, H_1)$

CMV_{90} = $(F_1, \text{“Postcondition Error..”}. java/lang/System.out, H_1)$

CMV_{92} = $(F_1, \mathcal{E}, H_1)$,

CMV_{95} = $(F_1, \mathcal{E}, H_1)$,

The constraint system generated from the current execution path is as follow:
$\boldsymbol{old\$balance_i = balance_i \wedge balance_i > 0 \wedge amount_i \geq 0 \wedge amount_i \leq balance_i \wedge amount_i < 100 \wedge balance_i = balance_i - amount_i * 25/100 \wedge balance_i\ != old\$balance_i - amount_i}$.

We observe that the constraints of this path are not conflicting. Indeed, the pre-state restricts the valid input values of this path to $\boldsymbol{amount_i \geq 0\ \ and\ amount_i < 100;}$ and the constraint $\boldsymbol{balance_i = balance_i - amout * 25/100}$ is matching with the non-postcondition $\boldsymbol{balance_i\ != old\$balance_i - amount_i}$. In fact, all the valid input values of the parameter amount included in the domain **[0, 100[** end in an invalid post-state. Therefore, we deduce that this execution path is not conform to its specification; as it begins in a state that is conform to the pre-state and does not satisfy the post-condition constraints. As consequence, the testing method is also not conform to the user specification.

6. Conclusion

This paper proposes a constraint-based verification approach for Java Bytecode programs augmented with its user specifications. We have showed firstly how we specify a java application at the Bytecode level using Static Bytecode Instrumentation. The main idea of the proposed work is to wrap the testing method with its specification expressed in pre/post conditions. We have illustrated secondly, how we can explore the memory constraint model deduced from java Bytecode method wrapped with Pre/Post conditions to detect a non-conformance between a given Java method and its specification. The main advantage of this approach is not only to detect the program inconsistencies relatively to their specifications, but also it shows exactly which execution path contains this inconsistency. Our work, is now oriented to detect path anomalies for secure testing.

References

[1] A. Bertolino, “Software testing research: Achievements, challenges, dreams”, In 2007 Future of Software Engineering, IEEE Computer Society, **(2007)**, pp. 85-103.
[2] K. Benlhachmi and M. Benattou, “A Formal Model of Conformity and Security Testing of Inheritance for Object Oriented Constraint Programming”, Journal of Information Security, vol. 4, no. 2, **(2013)**, pp. 113-123.
[3] F. Dadeau and F. Peureux, “Grey-box testing and verification of Java/JML”, Proc. - 4th IEEE Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2011, **(2011)**, pp. 298-303.
[4] R. DeMillo and J. Offut, “Constraint-based automatic test data generation”, IEEE Transactions on Software Engineering, vol. 17, no. 9, **(1991)**, pp. 900-910.
[5] J. Offut, Z. Jin and P. J., “The dynamic domain reduction procedure for test data generation”, Software–Practice and Experience, vol. 29, no. 2, **(1999)**, pp. 167-193.
[6] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado and W. E. Wong, “Establishing structural testing criteria for java bytecode”, Software Practice & Experience, vol. 36, no. 14, **(2006)**, pp. 1513-1541.
[7] J. Zhao, “Dependence analysis of Java bytecode”, In Computer Software and Applications Conference, COMPSAC 2000. The 24th Annual International, **(2000)**, pp. 486-491.

[8] A. Achkar, M. Benattou and J. L. Lanet, "Generating control flow graph from Java card byte code", In Information Science and Technology (CIST), 2014 Third IEEE International Colloquium in, **(2014)**, pp. 206-212.
[9] C. S. Pasareanu and N. Rungta, "Symbolic PathFinder: Symbolic execution of Java bytecode", Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10), **(2010)**, pp. 179-180.
[10] F. Charreteur and A. Gotlieb, "Constraint-based test input generation for Java bytecode", Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE, **(2010)**, pp. 131-140.
[11] F. Charreteur and A. Gotlieb, "Raisonnement contraintes pour le test de bytecode java", In quatrièmes Journées Francophones de Programmation par Contraintes (JFPC'08), Nantes, France, **(2008)**, pp. 11-20.
[12] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro and J. C. Maldonado, "JaBUTi: A Coverage Analysis Tool for Java Programs", In XVII SBES – Brazilian Symposium on Software Engineering (Tool Section), Manaus, AM, Brazil, **(2003)**, pp. 79-84.
[13] W. Xu, T. Ding and D. Xu, "Rule-Based Test Input Generation from Bytecode", In Software Security and Reliability, 2014 Eighth International Conference, **(2014)**, pp. 108-117.
[14] M. Benattou, J. Bruel, and N. Hameurlain, "Generating Test Data from OCL Specification", Proceedings of the ECOOP'2002 Work-Shop on Integration and Transformation of UML Models, **(2002)**, pp. 1-6.
[15] Y. Cheon and C. E. Rubio-Medrano, "Random test data generation for Java classes annotated with JML specifications", SERP, vol. 11, **(2007)**, pp. 385-392.
[16] E. Bernard, B. Legeard, X. Luck and F. Peureux. "Generation of test sequences from formal specifications: GSM 11-11 standard case study", International Journal of Software Practice and Experience, vol. 34, **(2004)**, pp. 915-948.
[17] Y. Cheon, "Automated random testing to detect specification-code inconsistencies", Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, Tech. Rep.07-07, **(2007)**.
[18] E. Bruneton, "ASM 4.0 A Java bytecode engineering library," http://download.forge.objectweb.org/asm/asm4-guide.pdf.
[19] J. Poole, "A Method to Determine a Basis Set of Paths to Perform Program Testing", Nat'l Inst. of Standards and Technology, **(1995)**.

