Formal Model of Conformity Analysis Method of Java Bytecode Programs annotated with BML Specifications

Safaa Achour, Mohammed Benattou, and Jean-Louis Lanet

Abstract—**This paper presents an analysis method of detecting the inconsistencies between a Java Bytecode program and its formal specification expressed at the Bytecode level, using the Bytecode Modeling Language (BML). The main objective of our work is not only to check the execution code of the called method of a Java object class from a valid input state but also to show how these inconsistencies can be extracted from the Control Flow Graph of the Method Under Test. We believe that Java Bytecode programs can contain bugs that the techniques used for objected-oriented software testing do not necessarily detect from their Java source programs. The proposed formal model shows how both the BML assertions and the Java Bytecode program can be translated into the same model of constraints, and how they can be used for detecting non-conformances of the method under test.**

Index Terms— **Bytecode Java, static testing, analysis, Bytecode Specifications, BML, Conformity Model, Constraint Model, JVM.**

I. Introduction

"TESTING by its nature can never conclude anything mathematically valid, as it amounts to taking a sample and trying to infer a generally valid judgment on the whole from the observed part"[1]. Software testing can be quantified and formalized when a strong basis for test generation can be defined [2]. In this sense, formal methods can play an essential role in software testing since they can precisely describe how the software is supposed to behave in a form that can be manipulated automatically.

Thus, the use of formal specifications to show conformance of an implementation to its specification has become a very used technique for software verification and can be performed either by randomly generated input test data, filtered via the precondition [3], or by using constraints resolution [4], [5]. Indeed, many methods use annotated programs together with techniques of constraint-solving to generate test cases [6] or to perform code-based verification of assertions. Such testing techniques are carried out in two steps: first, they translate the program to be verified and its specification into an intermediate representation, and from there, they extract the testing information.

Manuscript received March 17, 2020; revised January 07, 2021.

Safaa Achour is a researcher at the Computer Science Laboratory, Ibn Tofail University, Kénitra, Morocco. Email : safaa.achour@uit.ac.ma
Mohammed Benattou is a full professor of Computer Science Department, Ibn Tofail University, Morocco.
Email: mohammed.benattou@uit.ac.ma
Jean-Louis Lanet is a member of the Cidre research team Inria, Rennes, France. Email: jean-louis.lanet@inria.fr.

In object-oriented specification context, the Java Modeling Language (JML) [7] is an interface specification language for Java to, formally, specify the behavior of Java programs. One of the advantages of such annotation languages is that they allow expressing both model and code in the same file, sharing the same methods and class fields [4]. However, often these specification techniques are limited to Java source code, while for many cases, one needs to specify and verify the executable code of the application under test. Indeed, unlike the class files, which contain the Bytecode of Java programs, the source code is not always available; especially, for third party and mobile software. Furthermore, some critical applications are directly implemented at the executable level [12]. On the other hand, Java Bytecode programs can contain bugs that the methods used for object-oriented software testing do not necessarily detect from the Java source programs.

Many works have adapted white-box testing approaches to programs at lower level, either to extracting a control flow graph from the class file [8], [9], performing symbolic and concolic execution of Bytecode [10], or using constraint-based techniques to generate test inputs from the Bytecode of java programs [11]. However, those techniques alone do not guarantee that the called program behaves correctly regarding the user specification.

In this sense, the main idea of this paper is to extract the testing information from the Java Bytecode of the Method Under Test (MUT) and its formal specification expressed in the Bytecode Modeling Language (BML) [12], in the context of unit testing. Indeed, BML specification language is a variation of JML adapted to bytecode. It allows to specify the application at the Bytecode level. Furthermore, the source code JML predicates can be compiled into Bytecode BML predicates using the JML2BML compiler [12]. Hence, this allows us to test programs at the source code level, and still have the ability to perform Bytecode testing.

In [13], we have presented an example of generation of test cases for java Bytecode programs annotated with BML specifications. In this work, we firstly propose to formally express the BML method assertions coherently to the memory constraint model extracted from Java Bytecode Programs [11]. Secondly, we present a formal model of conformity of a given method using the constraint memory variables extracted from both the BML Specifications and from the Java Bytecode programs. Finally, we present code-based verification of assertions (static testing) for detecting non-conformances for a Method Under Test. In order to have the same model of constraints we propose to translate both the BML assertions and the Java Bytecode program into the same representation.

This paper is organized as follows: section 2 presents the related works, section 3 introduces some necessary preliminaries, section 4 describes the proposed formal constraint model of BML specifications, section 5 presents the formal model of conformity, section 6 describes an analysis method to illustrate the proposed approach, and finally section 7 gives some concluding remarks.

II. Related work

A. *Formal Specification-based Testing*

According to Binder [14], we cannot test without understanding what the software is supposed to do, and the software complexity may be represented by models to support test design. Thus, the formal specification can play an important role in software testing as they precisely define the functionalities offered by the System Under Test (SUT).

In an object-oriented context, the formal specification can be expressed as class invariant constraints characterizing the valid states of instances, and as method specifications describing the behavior of a method in terms of its preconditions and post-conditions [15]. Various testing methods use formal specification languages to show the conformance of an implementation to its specifications.

In [5], M. Benattou et al. propose a technique for test data generation based on OCL constraints using partition analysis of individual methods of a class; the idea is to reduce the given set of constraints into disjoint partitions using the Disjunctive Normal Form. In [16], the authors presented a constraint-based method on automated test generation from B models; they compute boundary states using a set constraint solver to build test cases by traversing the constrained reachability graph of the specification.

In [6], the proposed work shows how the application of constraint solving techniques to the verification and testing of Java/JML programs allows generating test data from the source code of the application under test, establishing code-based verification of assertions, and detecting the possible inconsistencies between a java program and its JML specification.

B. *Verification of the Java Bytecode Programs*

The principal reason to work at the Bytecode level is that we can extract structural testing information even when the source code is not available. Moreover, almost no information from the source code gets lost when compiling to Java Bytecode, so the program analysis and verification performed at this level can be reversed to the original high-level language through a reverse engineering process [17]. On the other hand, the Bytecode is free of compilation errors and optimized for execution. Thereby, the executable code of a given program allows us to have an idea about the structure of the code, which can help us to design better test cases.

Several works have been interested in structural testing either to perform code coverage[18][19][20], search testing[21][22][23][24], constraint based testing[22], [25], [26], or symbolic and dynamic execution[27][28][29][30]. Many of these structural techniques are adapted to programs at the Bytecode level. Indeed, in [8], the authors show how to generate an inter-procedural and intra-procedural control flow graph from the Bytecode of a given Java Card application. In [9], the authors present static and dynamic path executions of Bytecode programs using Control Flow Graph and Data Dependencies Graph. In [31], Vincenzi et al. presented Jabuti, a coverage testing tool designed to test and assess the quality of a given test set using the object code (Bytecode) of java applications and Java-based Components. In [10], a software analysis tool named Symbolic PathFinder (SPF) is described; SPF uses symbolic execution combined with model checking to automate the test case generation and error detection. The authors present in [32] a backward symbolic execution method for Java Bytecode programs; The idea behind this work is to reduce the simulation cases by deriving condition on inputs performing reverse execution for each Bytecode. In [33], they present an implementation of mutant generation at the Bytecode level and they support method-level mutant operators. In [34], Xu et al. show how to automate the generation of test inputs from Java bytecode by using a rule-based approach; this latter consists of using a set of predefined rules as search guidelines. In [11], the authors propose a goal-oriented method for automated generation of test inputs for Java Bytecode programs; They describe a new constraint memory model of the Java Virtual Machine (JVM), which allows backward exploration of the Bytecode program.

However, those verification techniques that perform analysis directly over the Bytecode, can only guarantee that the code is well-typed or well structured, or to generate input data that reach specific instruction in the program and consequently contribute to the reachability problem. Nevertheless, nothing guarantees that the System Under Test behaves as intended.

Some approaches dealt with this problem by adapting the Proof Carrying Code to the Java Bytecode programs annotated with Bytecode Modeling Language [12],[35],[36]. They allow the client to verify functional or security properties about the application via a formal proof that accompanies the executable code of the application [37], [38]. Yet, we think that even if establishing the proof obligations guarantees the absence of certain classes of errors, they are very expensive and difficult to be implemented.

C. *The proposed Idea*

The constraint model proposed by F.Charreteur et al.[11], contributes to the automation of the test data generation. Indeed, the main goal of their approach presented in [11] is the early detection of infeasible (non-executable) paths. However, their works do not consider the problem of a method called from an invalid state.

We believe that without the consideration of the information contained in the user specifications, nothing can help to detect the differences between the actual behavior of the implementation of a SUT and the expected behavior of its specifications

In this sense, the principal purpose of our proposed checking approach is to explore not only the information of the constraint model extracted from the Bytecode, but also to consider the information of the user specifications expressed in BML as it is shown in figure Fig.1. Indeed, the BML makes it possible to annotate and subsequently check the compliance of an application concerning its specification at the Bytecode level. On the other hand, the structure of the Bytecode program will help us to detect inconsistencies that are difficult to be found by using only the model of the application. Indeed, the latter does not provide sufficient coverage. Nevertheless, if one wants to verify Java Bytecode programs annotated with BML specifications, both the Java Bytecode of the program under test and the BML specifications have to be translated into an intermediate representation (i.e. the same system of constraints).

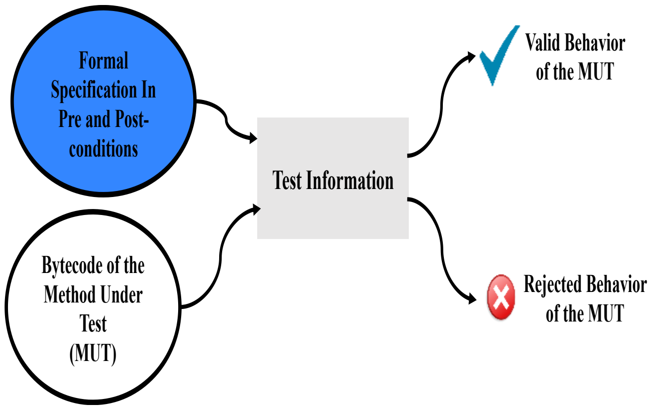

Formal Specification In Pre and Post-conditions
Bytecode of the Method Under Test (MUT)
Test Information
Valid Behavior of the MUT
Rejected Behavior of the MUT

Fig. 1. General Test Architecture.

To deal with this problem, we propose a formal model that transforms BML assertions into memory constraints coherently with the constraint model defined for Java Bytecodes. Indeed, as seen in the figure Fig. 2, we need to define a mapping of BML specifications into the constraint memory model representation [11] where the BML expressions and the evaluation of BML predicates are defined over the program JVM states (registers, operand stack, and heap).

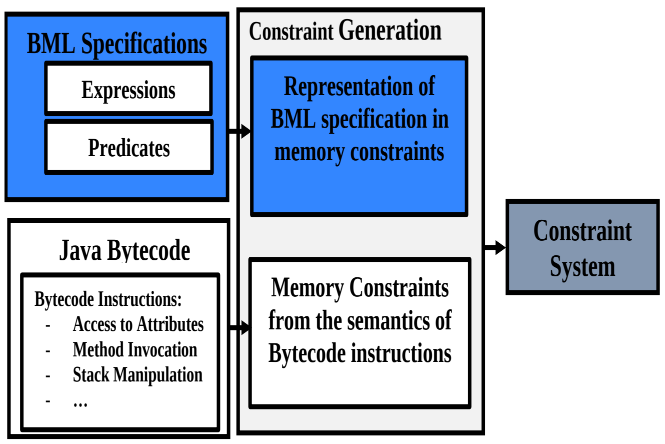


Fig. 2. Translation of Java Bytecode program annotated with BML specifications into constraints

Our idea is inspired by [6]; the main difference is that they perform test and verification at source level for Java/JML programs, while we work at the Bytecode level using the constraint memory model of JVM states.

In [35] and [36], the authors of BML language present a proof of correctness of Bytecode verification condition generator based on a weakest predicate transformer function. To do so, they have presented their proper model of Java Bytecode and they have given a formal meaning of BML language accordingly to it. The goal behind our work is to transform BML assertion into the bytecode memory model proposed by [11] to perform static testing (analysis), i.e. to detect the non-conforming paths of a given method, and consequently to detect the non-conforming methods.

In [39] and [40], we have proposed a testing methods for Java Bytecode programs instrumented with their specifications. However, the proposed methods presented the following disadvantages:

- The injection of the user specification in the application requires a good knowledge of Bytecode to be implemented.
- It becomes unworkable when the specification is complex.

III. Preliminaries

The ultimate objective of this work is to apply constraint-based testing on the Java Bytecode program annotated with BML. The principle is to transform the problem of generating the testing information, from the BML specifications and the Bytecode, into a problem of constraint solving.

In this section, we introduce some required preliminaries. We present firstly a preview of specification language tailored to Bytecode (BML: Bytecode Modeling Language); we also give a brief description of the Java Virtual Machine (JVM) representation and the memory constraint model defined for Java Bytecode instructions.

A. *Bytecode Modeling Language (BML)*

BML [12] is a Bytecode specification language that is designed to be closely related to JML (Java Modeling Language) [7], which is an interface specification language for Java to formally specify the behavior of Java programs.

1) *Overview of BML*

BML supports the essential features of JML. Thus, one can express the behavioral specification of Java Bytecode programs in the form of preconditions, post-conditions, and class invariant. Additionally, the JML source-level specifications can be compiled into BML Bytecode level specifications. In order to show the use of BML specifications, the example of the Java Bytecode specified with BML of the *"tranfer(Account dest, int amount)"* method of class *Account* is presented in figure Fig.3.

We note that the invariant of the class *Account* is stored in the class file as a special user-specific attribute [12]: Invariant: *#13 > 0*(which means that the attribute *balance* must always be positive).

Notice that the attribute *balance* that has been assigned the number *13* in the constant pool is not referenced with

Java source code of class Account annotated with JML Specifications

```
public class Account {

 //@invariant balance>0;
   private/*@spec_public@*/ int balance;

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

/*@requires amount>0;
  @ensures balance == \old(balance) + amount;
  @*/
public void deposit(int amount){
    balance =  balance + amount;
}

           ......


/*@requires amount>0 && amount < balance;
  @ensures balance == \old(balance) - amount
     && dest.balance = dest.balance + amount;
  @*/
public void transfer(Account dest, int amount){

   if (amount >= 2*this.balance/3 && amount <=
(3*this.balance)/4)
       this.balance = this.balance - amount * 25/100;
    else if(amount >= 0 && amount <balance)
       this.balance = this.balance - amount;
    else
       System.out.println("Invalid Operation");

 dest.balance = dest.balance + amount;
}

public int get() {
    return balance;
}
```

Java Bytecode of the method "transfer" annotated with BML Specification

```
public void transfer(Account, int);
{| requires lv[2] > 0 && lv[2] < lv[0].#13
   ensures lv[0].#13 = \old(lv[0].#13) – lv[2] &&      lv[1].#13
= \old(lv[1].#13) – lv[2] |}
    Code:
       0: iload_2
       1: iconst_2
       2: aload_0
       3: getfield      #13                 // Field balance:I
       6: imul
       7: iconst_3
       8: idiv
       9: if_icmplt     43
      12: iload_2
      13: iconst_3
      14: aload_0
      15: getfield      #13                 // Field balance:I
      18: imul
      19: iconst_4
      20: idiv
      21: if_icmpgt     43
      24: aload_0
      25: aload_0
      26: getfield      #13                 // Field balance:I
      29: iload_2
      30: bipush        25
      32: imul
      33: bipush        100
      35: idiv
      36: isub
      37: putfield      #13                 // Field balance:I
      40: goto          76
      43: iload_2
      44: iflt          68
      47: iload_2
      48: aload_0
      49: getfield      #13                 // Field balance:I
      52: if_icmpge     68
      55: aload_0
      56: aload_0
      57: getfield      #13                 // Field balance:I
      60: iload_2
      61: isub
      62: putfield      #13                 // Field balance:I
      65: goto          76
      68: getstatic     #24                 // Field
java/lang/System.out:Ljava/io/PrintStream;
      71: ldc           #30              // String Invalid Operation
      73: invokevirtual #32                 // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
      76: aload_1
      77: aload_1
      78: getfield      #13                 // Field balance:I
      81: iload_2
      82: iadd
      83: putfield      #13                 // Field balance:I
      86: return
```

Fig. 3. Example of Java/JML program and the Bytecode.

lv[0], as it is, implicitly, concerning all objects that are instance of the class ***Account***. Otherwise, it will always be explicitly referenced, for example, with *lv[0]* or *lv[1]* (*lv[0]* designate the reference This, and *lv[1]* designate the reference *dest*). On the other hand, *lv[2]* indicates the parameter *amount* of the method *transfer(Account dest, int amount)* stored in the register *2*.

2) Syntax of BML

In BML, only Bytecode expressions can be used. Therefore, all field names, class names, etc., are replaced by references to the constant pool (a number, preceded by the symbol #), whereas registers *lv* refer to parameters and local variables. The grammar also contains many specific keywords Bytecode, such as *cntr*: designating the stack counter; *st(e)* where *e* is an arithmetic expression referring to the eth element on the stack; and *length(a)* correspond to the length of array a. For more details, see section 3.1 of [12].

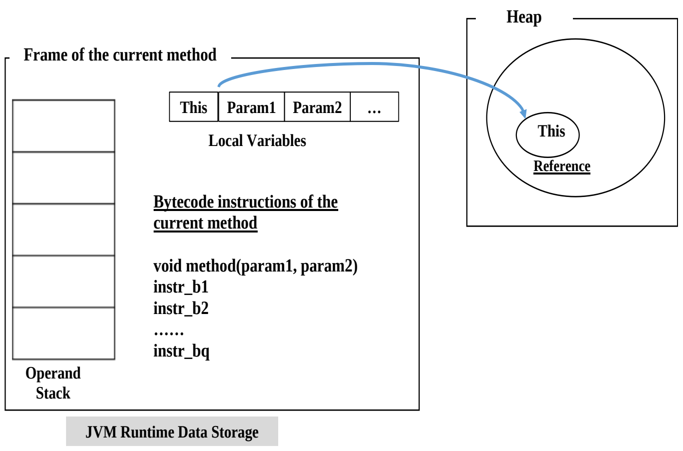


Fig. 4. Method execution in the JVM.

B. Constraint Memory Model

This sub-section gives a brief description of the Java Virtual Machine (JVM) representation, and the memory constraints on JVM states [11]. The memory model uses Constrained Memory Variables (CMV) to represent JVM states.

The JVM states represent the locations of runtime data storage; i.e. registers (local variables), 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, while the heap is the area of memory used by the JVM for the allocation of dynamic memory. Figure Fig.4 gives an illustration of runtime data storage of the method currently executed in the JVM.

The constraint-based testing of Java Bytecode programs requires the definition of a memory model [11]. This memory model is based on the notion of constrained memory variables CMV, where each Java Bytecode instruction of the program is seen as a relation between two CMVs: the CMV_k before the execution of this instruction, and the CMV_l after its execution.

A CMV is formally represented as a tuple (F, S, H) where F designates the set of registers, S designates the operand stack, and H represents the heap. Indeed, the registers are modeled by a function that associates a VTPR (the value contained in the register) to index i. The operand stack is modeled 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.

Moreover, the tuple (F, S, H) contains variables and domains. Integer and references are modeled by finite domain variables. On other hand, objects of the heap are modeled by pair elements: the type variable that represents the class of the object and the element associating an integer or reference variable to each attribute, which correspond to the value of the attribute.

IV. FORMAL CONSTRAINT MODEL OF BML

In this section, we give a formal transformation of BML assertions into constraints in the memory model. In this sense, we translate BML assertions (Expressions and predicates) into constraints over the registers, operand stack, or heap variables. We also present the validity of BML assertion in a given state CMV. Our work is based on axiomatic semantics concepts.

A. *Transformation state*

To extend the memory model with BML specification, we need to see BML assertions as constraints on JVM states. Particularly, the class invariant and the method specification will be considered as predicates that constrain the possible values of the content of the JVM memory. Indeed, the constraints contained in the precondition must be valid when the method is called; while the properties declared by the post-condition ensure that the called method has finished its execution correctly.

The class invariant represents constraints that every instance of the class must respect. The invariant also influences the satisfaction of method specification, as both the precondition and the post-condition are implicitly strengthened by the class invariants [41]. Therefore, the BML specifications constrain the state of registers and the state of the heap of the JVM in such a way that:

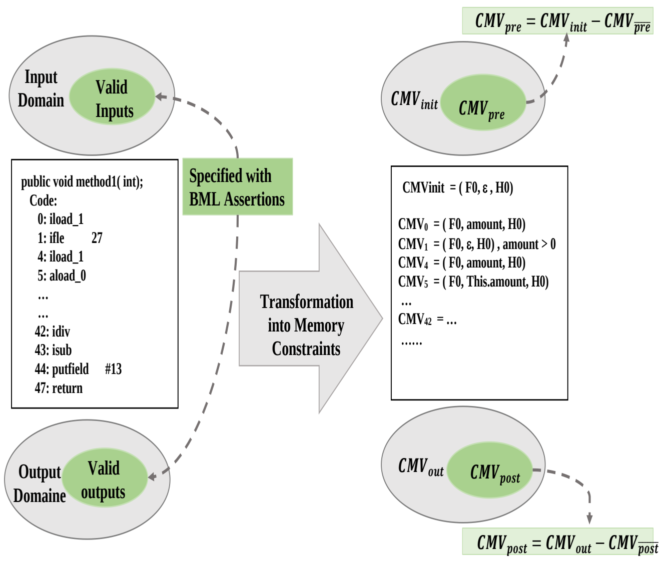
- The method parameters (integers or references type) located in registers F have to respect the precondition and the class invariant.
- The objects (and their attributes) that reside in the heap must respect the class invariant.
- The objects and parameters have to end the execution in a state that respects the post-condition.

We note that the stack is omitted here because it is empty at the beginning and the ending of program execution.

<u>*Definition 1*</u>: *A state CMV_{init} is defined as a valid state CMV_{pre} when it satisfies the constraints of BML precondition and class invariant.*

$$CMV_{init}=(F,\varepsilon,H)\xrightarrow{\mathit{Invariant} \wedge \mathit{Precondition}}CMV_{pre}=(F_{pre},\varepsilon,\,H_{pre})$$

We note that the symbol ε designates the empty stack.


$CMV_{pre} = CMV_{init} - CMV_{\overline{pre}}$
Input Domain
Valid Inputs
CMV_{init}
CMV_{pre}
Specified with BML Assertions
Transformation into Memory Constraints
Output Domaine
Valid outputs
CMV_{out}
CMV_{post}
$CMV_{post} = CMV_{out} - CMV_{\overline{post}}$

```
public void method1( int);
  Code:
    0: iload_1
    1: ifle       27
    4: iload_1
    5: aload_0
    ...
    ...
   42: idiv
   43: isub
   44: putfield     #13
   47: return
```

```
CMVinit  = ( F0, ε , H0)

CMV0  = ( F0, amount, H0)
CMV1  = ( F0, ε, H0) , amount > 0
CMV4  = ( F0, amount, H0)
CMV5  = ( F0, This.amount, H0)
...
CMV42  = ...
......
```

Fig. 5. Transformation of BML specification into memory constraints

F_{pre}: is the state of registers that allows only the valid values regarding the constraints required by the method precondition.

$$F_{pre}= F - \overline{\mathit{Precondition}}$$

H_{pre}: is the state of the heap that contains only the objects that respect the constraints of the class invariant.

$$H_{pre} = H - \overline{\mathit{inv}\,(O)}$$

<u>*Definition 2*</u>: *A final state CMV_{final} is defined as valid if it begins the execution from a valid initial state CMV_{pre} and ends in a state CMV_{post} that satisfy the BML postcondition and the class invariant constraints.*

Figure Fig.5 illustrates the translation of the BML precondition and post-condition into the constraint memory variable on the initial and the final state. Consequently, the constraints CMV_{pre} and CMV_{post} reduce the input and output domains to only valid values (relatively to the specification).

B. *Translation of BML expressions*

The fundamental step in the transformation of BML assertions is the translation of BML expressions in constraints coherently with the constraint memory model of the JVM. In this paper, we are particularly interested in the essential expressions that construct the BML pre- and post-conditions.

1) *Primitive and Reference Variables*

The values in BML assertions can be either a range of integers (byte, characters, short, Booleans and int) or a

TABLE I
BML SPECIFICATION AND ITS CORRESPONDENT CONSTRAINT MODEL

	BML Specifications	Constraint Model
Class invariant	**#13 > 0**	$\forall$ (Ref , balance) $\in$ H, Ref$\neq$ null, (i.e. This $\neq$ null$\wedge$dest$\neq$ null) balance $\in$ [0..INT_MAX]) (i.e. #13 > 0) where Ref can be any instance of the class Account
Precondition	lv[2] > 0 && lv[2] < lv[0].#13	lv[2]: 2$\rightarrow$ amount, This $\neq$ null, dest $\neq$ null amount >0 $\wedge$ amount <This.balance i.e. amount $\in$ [0.. This.balance]
Post-condition	lv[0].#13 = \old(lv[0].#13) – lv[2] && lv[1].#13 = \old(lv[1].#13) + lv[2]	This $\neq$ null, $(\text{This.balance})_{\text{post-state}} = (\text{This.balance})_{\text{pre\_state}}$ –amount $\wedge$ $(\text{dest.balance})_{\text{post-state}} = (\text{dest.balance})_{\text{pre\_state}}$+ amount

reference. As in the constraint model, integers and references are modeled by finite domain variables VTPR (designing Variable of Type Primitive or Reference).

$$(VTPR)_{CMV} = VTPR$$

2) Method parameters and local variables

In BML, the construct *lv(i)* denotes the local variables and the parameters located in the register of index *i*. Hence, their representation in the constraint model is made consequently. Therefore, the BML local variables *lv(i)* are modeled by a function that associates a value of primitive or reference type to the register of index *i*.$(lv(i))_{CMV}$:

$$i \rightarrow VTPR$$

3) Object attributes

The syntax of attribute access expressions in BML is E_{bml}.ident, the latter stands for the attribute value at index ident in the table of the constant pool for the reference designated by the expression E_{bml} [12][36].

The translation of $E_{bml}.ident$ to the constraint memory model is done by modeling this expression as a tuple where the first element is the object reference and the second is the value of the field referenced by the index ident:

$$(E_{bml}.ident)_{cmv} : E_{bml} \neq null,\ (E_{bml},\ [ident,\ VTPR]) \in H;$$

Since the reference E_{bml} has to access an attribute of index *ident*, it cannot be of a null value. The names of classes and attributes are used for simplification rather than their equivalent indexes in the table of constant pool as it is in BML

4) The operator "old"

A program expression *e* in an expression of the form *old(e)* refers to local variables (parameters) allocated in the pre-state CMV_{pre}. Indeed, The *old(e)* is a copy of the pre-state of a method execution generally used in the post-state CMV_{post}. Thus, its representation in the constraint model is as the following:

$old(e)_{CMVpost} = e_{CMVpre.}$

The expression *e* can be either a method parameter or an object attribute reachable from the parameters.

Example: Consider the BML Specifications of the method *transfer(Account, int)* presented in the figure Fig.3. The associated memory constraints to these specifications (class invariant, precondition, postcondition) are illustrated in the Table I.

C. BML Predicates

The translation process of the BML expressions into constraints in the memory model is now ended, and consequently, each predicate must be checked. The BML assertions can be seen as a first-order predicates logic, and they can be represented as predicates on the JVM memory state. Therefore, for the satisfiability of BML predicates to be performed, we need to precise the store (CMV in our case) and the values of logical variables that constitute the BML assertion. When each variable in an assertion is assigned a value (determined by the value of the program variables), the assertion becomes valid or invalid under a standard interpretation of a given predicate in a given state.

In this work, the interpretation of the BML predicates is defined over a program state CMV (i.e. the registers state, the operand stack state, and the heap state).

Definition 3:(Satisfiability of predicates)

Let P be a BML predicate and CMV = (f, s, h) a memory state of the JVM. The predicate P is valid in a state CMV (CMV ⊨ P) if the values of P are valid in the state CMV for any memory state referenced by P.

The satisfiability of an assertion (Predicate) is defined inductively as follows:

- *CMV ⊨P1 ∧ P2 if and only if CMV ⊨ P1 and CMV ⊨ P2*
- *CMV ⊨P1 ∨ P2 if and only if CMV ⊨ P1 or CMV ⊨ P2*
- *CMV ⊨ ¬ P1 if and only if not CMV ⊨ P1*
- *CMV ⊨ true is true in any state CMV*
- *CMV ⊨ false is false in any state CMV*
- *CMV ⊨ P1 = P2 if and only if < P1, CMV> ⊨ n1, <P2, CMV> ⊨ n2 and n1 = n2, with n1, n2 are respectively the values of P1 and P2 in a state CMV.*
- *CMV ⊨ P1 > P2 if and only if < P1, CMV> ⊨ n1, < P2, CMV> ⊨ n2 and n1 > n2, with n1, n2 are respectively the values of P1 and P2 in a state CMV.*
- *CMV ⊨ P1 < P2 if and only if < P1, CMV> ⊨ n1, < P2, CMV> ⊨ n2 and n1 < n2, with n1, n2 are*

respectively the values of P1 and P2 in a state CMV.

V. Formal Model of conformity

The combination of both the functional and the structural testing techniques allows us to test the behavior of a given application, and to know its internal working as well. However, the source code is not always available even more for third-party applications. In this sense, we propose to exploit first, the information contained in the BML specification, and secondly, the information contained in the Bytecode of the application (in the case of Java programs).

Indeed, the BML specification makes it possible to detect possible inconsistencies between the Bytecode program and its specification. On the other hand, the Bytecode of the program and its structure allows to test all parts of the programs, and to detect the paths that may contain these inconsistencies. In this section, we propose to check the validity of the method execution paths regarding the BML constraints. We also present our formal model of conformity of a given method using the constraint memory variables extracted from both the BML Specifications and from the Java Bytecode programs.

Testing Context: Let C be a class, and m be a method with n parameters $x = (x_1, x_2, \ldots, x_n)$. We define for each parameter x_i its domain of values E_i. We denote $E = E_1 \times E_2 \times \ldots \times E_n$ the domain of input vector of the method m. The control flow graph CFG is the internal representation of the method m. Let $P = \{p_1, p_2, \ldots, p_k\}$ be the set of all independent execution paths of the method m (extracted from the CFG).

Definition 4: (Valid paths relatively to the precondition)
An execution path pi of the set $P_{validprestate}$ is valid relatively to the method pre-state if pi satisfies both the precondition and the class invariant.

$$P_{validprestate} = P - P_{invalidprestate}$$

With $P_{invalidprestate}$ is the set of execution paths of the method m that do not satisfy the precondition or the invariant.

Definition 5:(Conform Execution Path)
A path from the set $P_{validprestate}$ is conform to the method specification if it terminates in state CMV_{post} that satisfy the post-condition (Post) and the invariant (Inv)

A path p_i of Method m is conform to its specification $\Leftrightarrow$
$\forall CMV_{pre}\ ((CMV_{pre} \models Pre \wedge Inv) \wedge <CMV_{pre} \wedge CMV_1 \wedge CMV_2 \wedge .. \wedge CMV_j> \downarrow CMV_{post}) \Rightarrow CMV_{post} \models post \wedge Inv$

Where $CMV_1, \ldots, CMV_j$ are the corresponding constraint memory variables of a path p_i, with j is the number of instructions in this path; and where CMV_{pre} and CMV_{post} respectively corresponding to the constraints extracted from the valid method pre-states and the valid post-states of BML assertions.

Indeed, a path p_i is conform to the method specification if it begins in a valid state CMV_{pre} that respects both the class invariant and the precondition constraints- and terminates in the state CMV_{post} that satisfy the post-condition *Post* and the invariant Inv.

Definition 6: (Conforming Method)
A method m is conform to its specification if all its valid execution paths are conform to the specification.

$\forall p_i \in P_{validprestate}$, *$p_i$ is conform to the method specification*
$\Rightarrow$ *Method m is conform to its specification.*

Consequently, a non-conform method is defined as follows:

Definition 7: m is a non-conform method relatively to its specification if: $p_i \in P_{validprestate}$ in such a way that p_i is not conforming to the method specification.

In other words, the method m does not conform to the BML Specification if:
$\exists CMV_{pre}$, $(CMV_{pre} \models pre \wedge Inv)$, $<CMV_{pre}, CMV_1, \ldots, CMV_j> \downarrow CMV_{\neg post} \Rightarrow CMV_{\neg post} \models \neg post \vee \neg Inv$

VI. Example of application

A. An Analysis Method: Detection of non-conformance (relatively to the postcondition)

In the context of this work, we present an analysis method (static testing method) that aims to detect the possible non-conformances between the Java Bytecode of a given method of a given class and its BML specification.

We consider that the Java Bytecode method of any class under test can be represented with a control flow graph (CFG) which gives a global overview of the execution paths that the input data can traverse during the execution. The set of the independent execution paths of the testing method is extracted using a basis path coverage method [42] based on the Depth First Search algorithm. Each one of these execution paths augmented with the BML precondition, the class invariant, and the negation of the post-condition is translated to the constraint memory system. This latter is deduced from the semantics of its java bytecode instructions and the semantics of BML contracts.

$$(CMV_{Pre} \wedge CMV_{Inv}) \wedge CMV_1 \wedge \ldots \wedge CMV_n \wedge (\neg CMV_{Post})$$

Where $CMV_1 \wedge \ldots \wedge CMV_n$ are the constraints generated from an execution path of the control flow graph, and CMV_{Pre}, CMV_{Inv}, $\neg CMV_{Post}$ are the generated constraints from BML specification.

The verification process will accumulate and check the consistency of these constraints. If the generated constraints are not contradictory, then this path contains an inconsistency relatively to the post-condition. Therefore, a non-conformance will be detected in the MUT. We note that the consistency of the constraints can be checked on the fly similarly to [11]. The principal advantage of this technique is that we can know precisely the paths where the detected inconstancies reside, even when the source code is not

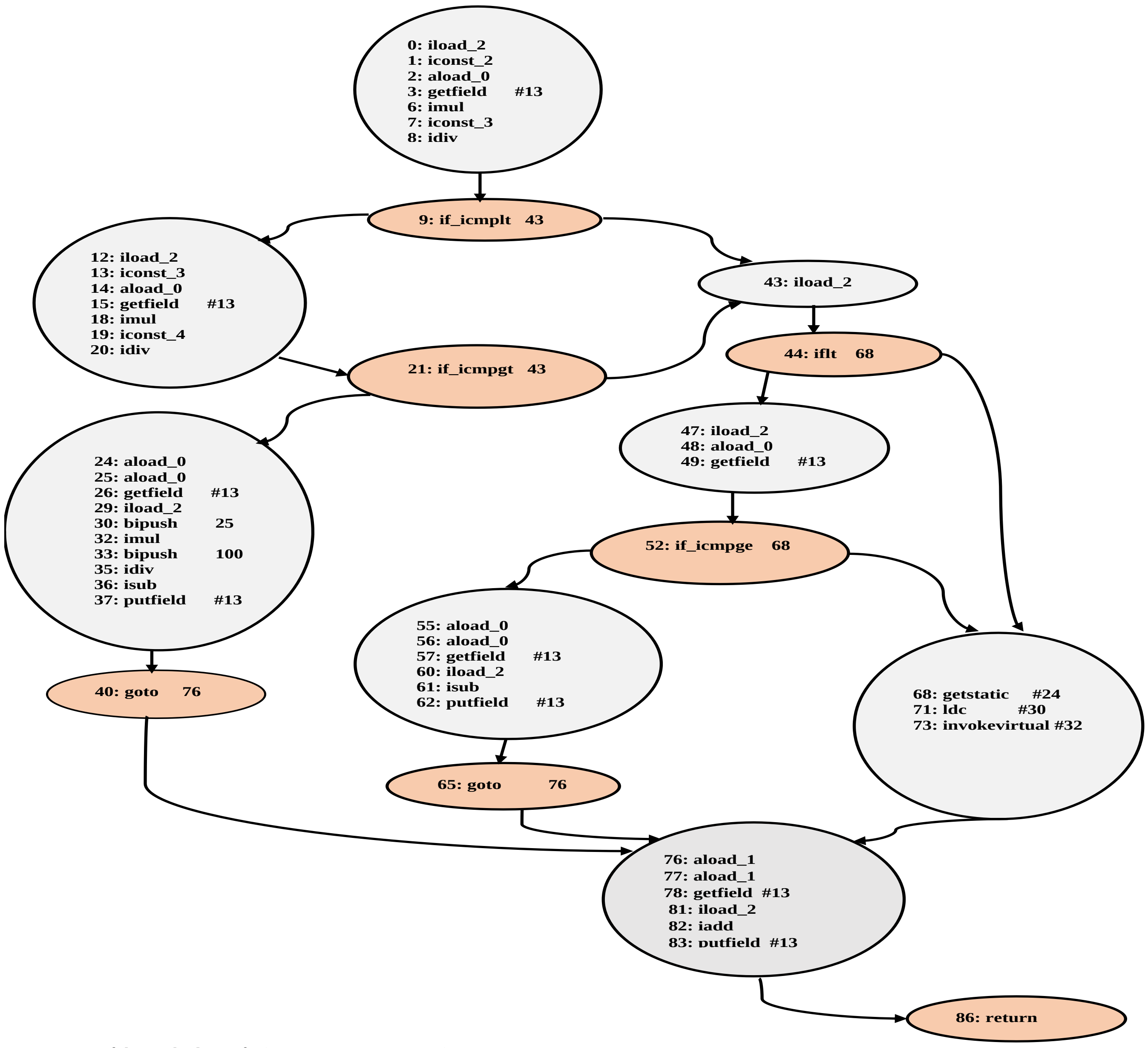


Fig. 6. CFG of the method transfer(Account acc, int amount)

available.

B. *Illustation*

Consider the Java Bytecode Method *transfer(Account dest, int amount)* annotated with BML specifications, shown in the figure Fig.3, and its control flow graph presented in the figure Fig. 6. Our main objective is to detect the non-conformances in the execution paths of the method under test.

The basis set of paths (all independent paths) of the MUT extracted from the CFG is represented as following:

Path 1 : 0 – 1 – 2 – 3 – 6 – 7 – 8 – 9 – 43 – 44 – 47 – 48 – 49 – 52 – 55 – 56 – 57 – 60 – 61 – 62 – 65 – 76 – 77 – 78 – 81 – 82 – 83 – 86.

Path 2 : 0 – 1 – 2 – 3 – 6 – 7 – 8 – 9 – 43 – 44 – 47 – 48 – 49 – 52 – 68 – 71 – 73 – 76 - 77 – 78 – 81 – 82 – 83 – 86.

Path 3: 0 – 1 – 2 – 3 – 6 – 7 – 8 – 9 – 12 – 13 – 14 – 15 – 18 – 19 – 20 – 21 – 24 – 25 – 26 – 29 – 30 – 32 – 33 – 35 – 36 – 37 – 40 – 76 – 77 – 78 – 81 – 82 – 83 – 86.

Path 4: 0 – 1 – 2 – 3 – 6 – 7 – 8 – 9 – 12 – 13 – 14 – 15 – 18 – 19 – 20 – 21 – 43 – 44 – 47 – 48 – 49 – 52 – 55 – 56 – 57 – 60 – 61 – 62 – 65 – 76 – 77 – 78 – 81 – 82 – 83 – 86.

Path 5: 0 – 1 – 2 – 3 – 6 – 7 – 8 – 9 – 12 – 13 – 14 – 15 – 18 – 19 – 20 – 21 – 43 – 44 – 47 – 48 – 49 – 52 – 68 – 71 – 73 – 76 - 77 – 78 – 81 – 82 – 83 – 86.

In order, to apply the analysis approach for detecting the possible non-conformances of the method *transfer(Account dest, int amount)* regarding the BML specifications, we firstly describe the constraints generated from the method execution paths, the method valid pre-state and the negation of the post-condition. Then, if these constraints are consistent, a non-conformance is detected.

The initial state: CMV_{init}= *($F_{0,}\varepsilon$, H_0)*. As mentioned before, the symbol ε designates the empty stack.

F_0= {0 →This,1→dest,2→amount}

F_0 is the set of registers: The reference *This (This refers to the current object)* is stored in the register number 0 (because the method *transfer* is not static), the register 1 contains the reference *dest*, whereas the register 2 stores the second parameter *amount*.

- **Path 1 :**

CMV_{pre}:∀(ref, balance) ∈ H, ref ≠ null, balance >0 amount> 0, This ≠null, amount <This.balance

CMV_0 = (F_0, amount, H_0)
CMV_1 = (F_0, 2.amount, H_0)
CMV_2 = (F_0, This.2.amount, H_0)
CMV_3 = (F_0, balance.2.amount, H_0); **This ≠ null, getfield(H_0, 13, This, balance)**
CMV_6 = (F_0, MUL.amount, H_0); **MUL = 2*balance**
CMV_7 = (F_0, 3.MUL.amount, H_0)
CMV_8 = (F_0, DIV.amount, H_0), **DIV = MUL /3**
CMV_9 = (F_0, ε, H_0), **amount < DIV**
CMV_{43} = (F_0, amount, H_0)
CMV_{44} = (F_0, ε, H_0), **amount > 0**
CMV_{47} = (F_0, amount, H_0)
CMV_{48} = (F_0, This.amount, H_0)
CMV_{49} = (F_0, balance.amount, H_0), **This ≠ null**, **getfield(H_0, 13, This, balance)**
CMV_{52} = (F_0, ε, H_0), **amount < balance**
CMV_{55} = (F_0, This, H_0),
CMV_{56} = (F_0, This.This, H_0),
CMV_{57} = (F_0, balance.This, H_0), This ≠ null, **getfield(H_0, 13, This, balance)**
CMV_{60} = (F_0, amount.balance.This, H_0),
CMV_{61} = (F_0, SUB.This, H_0), **SUB = balance – amount**
CMV_{62} = (F_0, ε , H_0), **This ≠ null, putfield(H_0, H_1, 13, This, SUB)**
CMV_{65} = (F_0, ε , H_0),
CMV_{76} = (F_0, dest , H_1);
CMV_{77} = (F_0, dest.dest , H_1);
CMV_{78} = (F_0, balance.dest , H_1); **dest≠ null, getfield(H_0, 13, dest, balance)**
CMV_{81} = (F_0, amount.balance.dest , H_1)
CMV_{82} = (F_0, ADD.balance.dest , H_1)
CMV_{83} = (F_0, ε, H_2); **dest≠ null, putfield(H_0, H_1, 13, dest, ADD)**
CMV_{86} = (F_0, ε, H_2);

$CMV_{\neg post}$:This.balance≠This.balance – amount ∨dest.balance≠dest.balance + amount

Therefore, the generated system of constraints of the execution path and the BML constraints (the method precondition and the negation of its post-condition) is the following; for simplification, we replace the generated memory constraint *putfield(...)*, which changes the state of the heap, by the symbol of affectation \$:=\$:

(∀(ref, balance) ∈ H, ref ≠ null, balance >0 ∧ amount > 0 ∧ This ≠null ∧ amount <This.balance)$_{pre}$∧ *amount > 0 ∧ amount < (2 * balance)/3 ∧ balance := balance – amount)* ∧dest≠ null∧*dest.balance = dest.balance + amount*∧ **(This.balance≠This.balance$_{old}$ – amount ∨dest.balance≠dest.balance$_{old}$ + amount)$_{\neg post}$**

After some simplifications, the constraint system reduces to the following:

(∀(ref, balance) ∈ H, ref ≠ null, balance >0 ∧ amount > 0 ∧*amount < (2 * balance)/3 ∧ balance := balance – amount)* ∧dest≠ null∧*dest.balance = dest.balance + amount*∧**This.balance≠This.balance$_{old}$ – amount)$_{\neg post}$**
∨
(∀(ref, balance) ∈ H, ref ≠ null, balance >0 ∧ amount > 0 ∧*amount < (2 * balance)/3 ∧balance := balance – amount)* ∧dest≠ null∧*dest.balance = dest.balance + amount*∧ **(dest.balance≠dest.balance$_{old}$ + amount)$_{\neg post}$**

- **Path 2**

CMV_{pre}:∀(ref, balance) ∈ H, ref ≠ null, balance >0 amount> 0,This ≠null, amount <This.balance

CMV_0 = (F_0, amount, H_0)
CMV_1 = (F_0, 2.amount, H_0)
CMV_2 = (F_0, This.2.amount, H_0)
CMV_3 = (F_0, balance.2.amount, H_0); **This ≠ null,getfield(H_0, 13, This, balance)**
CMV_6 = (F_0, MUL.amount, H_0); **MUL = 2*balance**
CMV_7 = (F_0, 3.MUL.amount, H_0)
CMV_8 = (F_0, DIV.amount, H_0), **DIV = MUL /3**
CMV_9 = (F_0, ε, H_0), **amount < DIV**
CMV_{43} = (F_0, amount, H_0)
CMV_{44} = (F_0, ε, H_0), **amount < 0**
…

$CMV_{\neg post}$:This.balance≠This.balance – amount ∨dest.balance≠dest.balance + amount
After simplifications, the constraint system is the following
∀(ref, balance) ∈ H ∧ ref ≠ null ∧ balance >0∧amount > 0 ∧ This ≠null ∧ amount <This.balance∧amount < 0 ∧ …

- **Path 3**

CMV_{pre}:∀(ref, balance) ∈ H, ref ≠ null, balance >0 amount> 0, This ≠null, amount <This.balance

CMV_0 = (F_0, amount, H_0)
CMV_1 = (F_0, 2.amount, H_0)
CMV_2 = (F_0, This.2.amount, H_0)
CMV_3 = (F_0, balance.2.amount, H_0); **This ≠ null,getfield(H_0, 13, This, balance)**
CMV_6 = (F_0, MUL.amount, H_0); **MUL = 2*balance**
CMV_7 = (F_0, 3.MUL.amount, H_0)
CMV_8 = (F_0, DIV.amount, H_0), **DIV = MUL /3**
CMV_9 = (F_0, ε, H_0), **amount ≥ DIV**

TABLE II
THE GENERATED CONSTRAINTS FOR EACH INDEPENDENT PATH OF TRANSFER METHOD

Path Number	Generated Constraint System from ($pre_{bml} \wedge Inv_{bml} \wedge Path\_i \wedge \neg Post_{bml}$)	Decision
Path 1	$(\forall(ref, balance) \in H, ref \neq null, balance > 0 \wedge amount > 0 \wedge amount < (2 * balance)/3 \wedge balance := balance - amount) \wedge dest \neq null \wedge dest.balance = dest.balance + amount \wedge This.balance \neq This.balance_{old} - amount)_{\neg post}$ $\vee$ $(\forall(ref, balance) \in H, ref \neq null, balance > 0 \wedge amount > 0 \wedge amount < (2 * balance)/3 \wedge balance := balance - amount) \wedge dest \neq null \wedge dest.balance = dest.balance + amount \wedge (dest.balance \neq dest.balance_{old} + amount)_{\neg post}$	Eliminated (Contradictory Constraints)
Path 2	$\forall(ref, balance) \in H \wedge ref \neq null \wedge balance > 0 \wedge amount > 0 \wedge This \neq null \wedge amount < This.balance \wedge amount < 0 \wedge \ldots$	Eliminated (Contradictory Constraints, Especially between the path constraints and the precondition)
Path 3	$\forall(ref, balance) \in H, ref \neq null, balance > 0 \wedge amount > 0 \wedge This \neq null \wedge amount \geq (2 * balance)/3 \wedge amount \leq 3*balance/4 \wedge balance := balance - (25 * amount/100) \wedge dest \neq null \wedge dest.balance = dest.balance + amount \wedge This.balance \neq This.balance_{old} - amount)_{\neg post}$ $\vee$ ………	**Consistent Constraints** **(Non-Conformance detected)**
Path 4	$\forall(ref, balance) \in H, ref \neq null, balance > 0 \wedge amount \geq (3*balance/4) \wedge amount < balance \wedge balance := balance - amount) \wedge dest \neq null \wedge dest.balance = dest.balance + amount \wedge This.balance \neq This.balance_{old} - amount)_{\neg post}$ $\vee$ $\forall(ref, balance) \in H, ref \neq null, balance > 0 \wedge amount \geq (3*balance/4) \wedge amount < balance \wedge balance := balance - amount) \wedge dest \neq null \wedge dest.balance = dest.balance + amount \wedge (dest.balance \neq dest.balance_{old} + amount)_{\neg post}$	Eliminated (Contradictory Constraints)
Path 5	$\forall(ref, balance) \in H, ref \neq null, balance > 0 \wedge amount > 0 \wedge This \neq null \wedge amount < This.balance \wedge amount \geq This.balance \wedge \ldots$	Eliminated (Contradictory Constraints, Especially between the path constraints and the precondition)

CMV_{12} = (F_0, amount, H_0),
CMV_{13} = (F_0, 3.amount, H_0)
CMV_{14} = (F_0, This.3.amount, H_0)
CMV_{15} = (F_0, balance.3.amount, H_0); **This ≠ null, getfield(H_0, 13, This, balance)**
CMV_{18} = (F_0, MUL2.amount, H_0), **MUL2 = 3*balance**
CMV_{19} = (F_0, 4.MUL2.amount, H_0)
CMV_{20} = (F_0, DIV2.amount, H_0); **DIV2 = MUL2/4**
CMV_{21} = (F_0, c, H_0), **amount ≤ DIV2**
CMV_{24} = (F_0, This, H_0),
CMV_{25} = (F_0, This.This, H_0),
CMV_{26} = (F_0, balance.This, H_0), **This ≠ null, getfield(H_0, 13, This, balance)**
CMV_{29} = (F_0, amount.balance.This, H_0),
CMV_{30} = (F_0, 25.amount.balance.This, H_0),
CMV_{32} = (F_0, MUL3.balance.This, H_0), **MUL3 = 25*amount**
CMV_{33} = (F_0, 100.MUL3.balance.This, H_0),
CMV_{35} = (F_0, DIV3.balance.This, H_0), **DIV3=MUL3 / 100**
CMV_{36} = (F_0, SUB.This, H_0), **SUB = balance – DIV3**
CMV_{37} = (F_0, ε , H_1); **This ≠ null, putfield(H_0, H_1, 13, This, SUB)**
CMV_{40} = (F_0, ε , H_1)
CMV_{76} = (F_0, dest , H_1);
CMV_{77} = (F_0, dest.dest , H_1);
CMV_{78} = (F_0, balance.dest , H_1); **dest≠ null, getfield(H_0, 13, dest, balance)**
CMV_{81} = (F_0, amount.balance.dest , H_1)
CMV_{82} = (F_0, ADD.balance.dest , H_1)
CMV_{83} = (F_0, ε, H_2); **dest≠ null, putfield(H_0, H_1, 13, dest, ADD)**
CMV_{86} = (F_0, ε, H_2);
$CMV_{\neg post}$:This.balance≠This.balance – amount ∨dest.balance≠dest.balance + amount

After some simplifications, the constraint system is as follow:

$\forall(ref, balance) \in H, ref \neq null, balance > 0 \wedge amount > 0 \wedge This \neq null \wedge amount \geq (2 * balance)/3 \wedge amount \leq 3*balance/4 \wedge balance := balance - (25 * amount/100) \wedge dest \neq null \wedge dest.balance = dest.balance + amount \wedge This.balance \neq This.balance_{old} - amount)_{\neg post}$

$\vee$

$\forall(ref, balance) \in H, ref \neq null, balance > 0 \wedge amount > 0 \wedge This \neq null \wedge amount \geq (2 * balance)/3 \wedge amount \leq 3*balance/4 \wedge balance := balance - (25 * amount)/100 \wedge dest \neq null \wedge dest.balance = dest.balance + amount \wedge dest.balance \neq dest.balance + amount)_{\neg post}$

- **Path4**

CMV_{pre}: ∀(ref, balance) ∈ H, ref ≠ null, balance >0 amount> 0, This ≠null, amount <This.balance
CMV_0 = (F_0, amount, H_0)
CMV_1 = (F_0, 2.amount, H_0)
CMV_2 = (F_0, This.2.amount, H_0)
CMV_3 = (F_0, balance.2.amount, H_0); **This ≠ null, getfield(H_0, 13, This, balance)**
CMV_6 = (F_0, MUL.amount, H_0); **MUL = 2*balance**
CMV_7 = (F_0, 3.MUL.amount, H_0)

CMV$_{8}$ = (F$_0$, DIV.amount, H$_0$), **DIV = MUL /3**
CMV$_{9}$ = (F$_0$, ε, H$_0$), **amount ≥ DIV**
CMV$_{12}$ = (F$_0$, amount, H$_0$),
CMV$_{13}$ = (F$_0$, 3.amount, H$_0$)
CMV$_{14}$ = (F$_0$, This.3.amount, H$_0$)
CMV$_{15}$ = (F$_0$, balance.3.amount, H$_0$); **This ≠ null, getfield(H$_0$, 13, This, balance)**
CMV$_{18}$ = (F$_0$, MUL2.amount, H$_0$), **MUL2 = 3*balance**
CMV$_{19}$ = (F$_0$, 4.MUL2.amount, H$_0$)
CMV$_{20}$ = (F$_0$, DIV2.amount, H$_0$); **DIV2 = MUL2/4**
CMV$_{21}$ = (F$_0$, c, H$_0$), **amount > DIV2**
CMV$_{43}$ = (F$_0$, amount, H$_0$)
CMV$_{44}$ = (F$_0$, ε, H$_0$), **amount > 0**
CMV$_{47}$ = (F$_0$, amount, H$_0$)
CMV$_{48}$ = (F$_0$, This.amount, H$_0$)
CMV$_{49}$ = (F$_0$, balance.amount, H$_0$), **This ≠ null, getfield(H$_0$, 13, This, balance)**
CMV$_{52}$ = (F$_0$, ε, H$_0$), **amount < balance**
CMV$_{55}$ = (F$_0$, This, H$_0$),
CMV$_{56}$ = (F$_0$, This.This, H$_0$),
CMV$_{57}$ = (F$_0$, balance.This, H$_0$), **This ≠ null, getfield(H$_0$, 13, This, balance)**
CMV$_{60}$ = (F$_0$, amount.balance.This, H$_0$),
CMV$_{61}$ = (F$_0$, SUB.This, H$_0$), **SUB = balance – amount**
CMV$_{62}$ = (F$_0$, ε , H$_0$), **This ≠ null, putfield(H$_0$, H$_1$, 13, This, SUB)**
CMV$_{65}$ = (F$_0$, ε , H$_0$),
CMV$_{76}$ = (F$_0$, dest , H$_1$);
CMV$_{77}$ = (F$_0$, dest.dest , H$_1$);
CMV$_{78}$ = (F$_0$, balance.dest , H$_1$); **dest≠ null, getfield(H$_0$, 13, dest, balance)**
CMV$_{81}$ = (F$_0$, amount.balance.dest , H$_1$)
CMV$_{82}$ = (F$_0$, ADD.balance.dest , H$_1$)
CMV$_{83}$ = (F$_0$, ε, H$_2$); **dest≠ null, putfield(H$_0$, H$_1$, 13, dest, ADD)**
CMV$_{86}$ = (F$_0$, ε, H$_2$);

CMV$_{\neg post}$:This.balance≠This.balance – amount ∨dest.balance≠dest.balance + amount

After some simplifications, the constraint system is the following:

∀(ref, balance) ∊ H, ref ≠ null, balance > 0 ∧ This ≠ null ∧ amount ≥ (3*balance/4) ∧ amount < balance *∧balance := balance – amount) ∧dest≠ null∧dest.balance = dest.balance + amount∧***This.balance≠This.balance$_{old}$ – amount)$_{\neg post}$**

∨

∀(ref, balance) ∊ H, ref ≠ null, balance > 0 ∧ amount ≥ (3*balance/4) ∧ This ≠ null ∧ amount < balance *∧balance := balance – amount) ∧dest≠ null∧dest.balance = dest.balance + amount∧* **(dest.balance≠dest.balance$_{old}$ + amount)$_{\neg post}$**

- **<u>Path 5</u>**

CMV$_{pre}$: ∀(ref, balance) ∊ H, ref ≠ null, balance >0 amount> 0, This ≠null, amount <This.balance

CMV$_{0}$ = (F$_0$, amount, H$_0$)
CMV$_{1}$ = (F$_0$, 2.amount, H$_0$)
CMV$_{2}$ = (F$_0$, This.2.amount, H$_0$)
CMV$_{3}$ = (F$_0$, balance.2.amount, H$_0$); **This ≠ null, getfield(H$_0$, 13, This, balance)**
CMV$_{6}$ = (F$_0$, MUL.amount, H$_0$); **MUL = 2*balance**
CMV$_{7}$ = (F$_0$, 3.MUL.amount, H$_0$)
CMV$_{8}$ = (F$_0$, DIV.amount, H$_0$), **DIV = MUL /3**
CMV$_{9}$ = (F$_0$, ε, H$_0$), **amount ≥ DIV**
CMV$_{12}$ = (F$_0$, amount, H$_0$),
CMV$_{13}$ = (F$_0$, 3.amount, H$_0$)
CMV$_{14}$ = (F$_0$, This.3.amount, H$_0$)
CMV$_{15}$ = (F$_0$, balance.3.amount, H$_0$); **This ≠ null, getfield(H$_0$, 13, This, balance)**
CMV$_{18}$ = (F$_0$, MUL2.amount, H$_0$), **MUL2 = 3*balance**
CMV$_{19}$ = (F$_0$, 4.MUL2.amount, H$_0$)
CMV$_{20}$ = (F$_0$, DIV2.amount, H$_0$); **DIV2 = MUL2/4**
CMV$_{21}$ = (F$_0$, c, H$_0$), **amount > DIV2**
CMV$_{43}$ = (F$_0$, amount, H$_0$)
CMV$_{44}$ = (F$_0$, ε, H$_0$), **amount > 0**
CMV$_{47}$ = (F$_0$, amount, H$_0$)
CMV$_{48}$ = (F$_0$, This.amount, H$_0$)
CMV$_{49}$ = (F$_0$, balance.amount, H$_0$), **This ≠ null, getfield(H$_0$, 13, This, balance)**
CMV$_{52}$ = (F$_0$, ε, H$_0$), **amount ≥ balance**
….
CMV$_{\neg post}$:This.balance≠This.balance – amount ∨dest.balance≠dest.balance + amount

After some simplifications, the constraint system is as follow:

(∀(ref, balance) ∊ H, ref ≠ null, balance >0∧ amount > 0 ∧ This ≠null ∧ amount <This.balance∧amount ≥ This.balance∧ …

TABLE III
THE EXECUTION PATHS OF TRANSFER METHOD AND THEIR RELATIVE VALID INPUTS PARTITIONS

Path Number	Valid Input Sub-Domain	Non-Conformance Detected
Path 1	**amount ∊]0,(2*balance)/3[**∧ balance ∊]0,INT_MAX[∧ This ≠ null ∧dest≠ null	No
Path 2	Invalid pre-state	No
Path 3	**amount ∊ [(2*balance)/3 , (3*balance)/4]**∧ balance ∊]0,INT_MAX[∧ This ≠ null ∧dest≠ null	**Yes**
Path 4	**amount ∊](3*balance)/4 , balance]**∧ balance ∊]0,INT_MAX[∧ This ≠ null ∧dest≠ null	No
Path 5	Invalid pre-state	No

<u>Constraints Analysis</u>

The Table II presents the constraint system generated from each execution path of the method *transfer(Account dest, int amount)*.

Effectively, in the two paths, *Path 1* and *Path 4*, the generated constraints are contradictory; this means that these two paths do not contain any inconsistencies (non-conformances) with the specification. Therefore, those paths are discarded.

In the *Path 3*, the constraints generated from the precondition, the invariant, the execution path, and the negation of the post-condition are consistent. Therefore, this path contains a non-conformance regarding the specification.

The paths *Path 2* and *Path 5* will also be discarded due to the conflicting constraint in the execution path and the precondition constraints. In other words, the inputs traversing this path do not respect the precondition in the first place.

As it is seen in Table III, we also observe, that the method precondition allows us to restrict the input domain to valid data. Whereas, the path constraints help us to divide the valid input domain into sub-domains (partitions), where each partition represents the input data that can traverse this sub-domain.

As our objective is the detection of non-conformances in the method transfer relatively to its BML specification, we can see from Tables II and III that in the sub-domain (**amount $\in$ [(2*balance) / 3 , (3*balance)/4]**$\wedge$ balance $\in$]0,INT_MAX[$\wedge$ This $\neq$ null $\wedge$dest$\neq$ null), the path 3 does not respect the specification. Consequently, the method transfer is non-conforming to its specification.

Indeed,

$\exists CMV_{pre}$, $(CMV_{pre} \vDash$(**amount $\in$]0, balance [**$\wedge$ balance $\in$]0,INT_MAX[$\wedge$ This $\neq$ null $\wedge$dest$\neq$ null), <$\forall$**(ref, balance) $\in$ H, ref $\neq$ null, balance >0 $\wedge$ amount > 0 $\wedge$ This $\neq$null $\wedge$ amount $\geq$ (2 * balance)/3 $\wedge$ amount $\leq$ 3*balance/4 $\wedge$ balance = balance –(25 * amount/100)** >$\swarrow$ $CMV_{\neg post} \Rightarrow CMV_{\neg post} \vDash$ *(***This.balance$\neq$This.balance$_{old}$ – amount)$_{\neg post}$**

Testing a program from the specification or the model solely [4], [5], [15], [43] allows us to test the behavior of programs, but since the latter is a black box, we cannot know the covered paths or the tested part of the application. For instance, with the specification alone, it will be difficult to find the non-conformance detected in the previous example. In this sense, the information of the Bytecode facilitates the detection of the program parts that lead to non-conform behaviors.

VII. CONCLUSION

This paper proposes a code-based analysis of assertions for Java Bytecode programs annotated with BML. Most of the existing works, about the test or the analysis of Java Bytecode programs, do not consider the case of programs called from an invalid input state. The main objective of our work is, on one hand, to check if the behavior rejected by a specification, expressed formally, is rejected by its implementation as well. On the other hand, the Bytecode of the testing method gives the structure of the program even when the source code is not available. The proposed method allows detecting which paths may contain the detected inconsistencies. We have firstly presented a formal model of transforming BML assertions into constraints on JVM memory states. Secondly, we have proposed a formal model of the conformity of an execution path of a given method. Finally, we have presented an analysis method of detecting the inconsistencies between a Java Bytecode Method and its BML Specifications. Our future work is now oriented to test the Bytecode of Java method called from invalid data.

REFERENCES

[1] A. Bertolino, « Software testing research and practice », in International Workshop on Abstract State Machines, 2003, p. 1–21.

[2] A. J. Offutt, Y. Xiong, et S. Liu, « Criteria for generating specification-based tests », in *Proceedings Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)(Cat. No. PR00434)*, 1999, p. 119–129.

[3] Y. Cheonet C. E. Rubio-Medrano, « Random test data generation for Java classes annotated with JML specifications », 2007.

[4] F. Bouquet, F. Dadeau, et B. Legeard, « Automated boundary test generation from JML specifications », in *International Symposium on Formal Methods*, 2006, p. 428–443.

[5] M. Benattou, J.-M. Bruel, et N. Hameurlain, « Generating test data from OCL specification », 2002.

[6] F. Dadeau et F. Peureux, « Grey-box testing and verification of Java/JML », in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, p. 298–303.

[7] G. T. Leavens, A. L. Baker, et C. Ruby, « JML: a Java modeling language », in *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, 1998, p. 404–420.

[8] A. Amine, B. Mohammed, et L. Jean-Louis, « Generating control flow graph from Java card byte code », in *2014 Third IEEE International Colloquium in Information Science and Technology (CIST)*, 2014, p. 206–212.

[9] S. Soomro, Z. Alansari, et M. R. Belgaum, « Path executions of java bytecode programs », in *Progress in Advanced Computing and Intelligent Engineering*, Springer, 2018, p. 261–271.

[10] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, et N. Rungta, « Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis », *Automated Software Engineering*, vol. 20, nº 3, p. 391–425, 2013.

[11] F. Charreteuret A. Gotlieb, « Constraint-based test input generation for java bytecode », in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010, p. 131–140.

[12] L. Burdy, M. Huisman, et M. Pavlova, « Preliminary design of BML: A behavioral interface specification language for Java bytecode », in *International Conference on Fundamental Approaches to Software Engineering*, 2007, p. 215–229.

[13] S. Achour et M. Benattou, « Test case generation for Java Bytecode programs annotated with BML specifications », in *2016 5th International Conference on Multimedia Computing and Systems (ICMCS)*, p. 605–610.

[14] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[15] K. Benlhachmiet M. Benattou, « A Formal Model of Conformity and Security Testing of Inheritance for Object Oriented Constraint Programming », 2013.

[16] E. Bernard, B. Legeard, X. Luck, et F. Peureux, « Generation of test sequences from formal specifications: GSM 11-11 standard case study », *Software: Practice and Experience*, vol. 34, nº 10, p. 915–948, 2004.

[17] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, et W. E. Wong, « Establishing structural testing criteria for java bytecode », *Software: practice and experience*, vol. 36, nº 14, p. 1513–1541, 2006.

[18] L. Lun, X. Chi, et H. Xu, "Coverage Criteria for Component Path-oriented in Software Architecture", *Engineering Letters*, vol.27, nº 1, pp 40–52, 2019.

[19] R. Gopinath, C. Jensen, et A. Groce, « Code coverage for suite evaluation by developers », in *Proceedings of the 36th International Conference on Software Engineering*, 2014, p. 72–82.

[20] B. Lesage, S. Law, et I. Bate, « TACO: An industrial case study of Test Automation for COverage », in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, 2018, p. 114–124.

[21] G. Fraser et A. Arcuri, « Evolutionary Generation of Whole Test Suites », in *2011 11th International Conference on Quality Software*, juill. 2011, p. 31-40, doi: 10.1109/QSIC.2011.19.

[22] J. Malburget G. Fraser, « Combining search-based and constraint-based testing », in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, p. 436–439.

[23] A. Sharma, P. Rishon, et A. Aggarwal, « Software testing using genetic algorithms », *Int. J. Comput. Sci. Eng. Surv.(IJCSES)*, vol. 7, nº 2, p. 21–33, 2016.

[24] I. T. Elgendy, M. R. Girgis, et A. A. Sewisy, "A GA-Based Approach to Automatic Test Data Generation for ASP .NET Web Applications", *IAENG International Journal of Computer Science*, vol. 47, nº 3, pp 557-564, 2020.

[25] L. Al Sardy, F. Saglietti, T. Tang, et H. Sonnenberg, « Constraint-based testing for buffer overflows », in *International Conference on Computer Safety, Reliability, and Security*, 2018, p. 99–111.

[26] A. Gotlieb, « Euclide: A constraint-based testing framework for critical c programs », in *2009 International Conference on Software Testing Verification and Validation*, 2009, p. 151–160.

[27] W. Visser, C. S. Păsăreanu, et S. Khurshid, « Test input generation with Java PathFinder », in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004, p. 97–107.

[28] C. Cadaret K. Sen, « Symbolic execution for software testing: three decades later », *Communications of the ACM*, vol. 56, nº 2, p. 82–90, 2013.

[29] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, et I. Finocchi, « A survey of symbolic execution techniques », *ACM Computing Surveys (CSUR)*, vol. 51, nº 3, p. 1–39, 2018.

[30] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, et Y. Lin, « Towards optimal concolic testing », in *Proceedings of the 40th International Conference on Software Engineering*, 2018, p. 291–302.

[31] A. M. Vincenzi, M. E. Delamaro, et J. C. Maldonado, « JaBUTi–Java Bytecode Understanding and Testing », *São Carlos, SP: Universidade de São Paulo–UPS*, 2003.

[32] T. Inafune, S. Miura, T. Taketa, et Y. Hiranaka, « Symbolic backward simulation of Java bytecode program », in *Proceedings of the 10th International Conference on Computer Modeling and Simulation*, 2018, p. 140–145.

[33] H. Coles, T. Laurent, C. Henard, M. Papadakis, et A. Ventresque, « Pit: a practical mutation testing tool for java », in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, p. 449–452.

[34] W. Xu, T. Ding, et D. Xu, « Rule-Based Test Input Generation from Bytecode », in *2014 Eighth International Conference on Software Security and Reliability (SERE)*, 2014, p. 108–117.

[35] M. Pavlova, « Vérification de bytecode et ses application », PhD Thesis, École Supérieure en Sciences Informatiques de Sophia Antipolis, 2007.

[36] M. Pavlova, " Bytecode Verification and its Applications ", Technical drafts, École Supérieure en Sciences Informatiques de Sophia Antipolis.

[37] L. Burdyet M. Pavlova, « Java bytecode specification and verification », in *Proceedings of the 2006 ACM symposium on Applied computing*, 2006, p. 1835–1839.

[38] G. Barthe*et al.*, « JACK—a tool for validation of security and behaviour of Java applications », in *International Symposium on Formal Methods for Components and Objects*, 2006, p. 152–174.

[39] S. Achour et M. Benattou, « A Model Based Testing Approach for Java Bytecode Programs. », *JCP*, vol. 13, nº 9, p. 1098–1114, 2018.

[40] S. Achour, A. Chouenyib, et M. Benattou, « A Constraint-Based Verification Approach for Java Bytecode Programs », *International Journal of Software Engineering and Its Applications*, vol. 12, nº 2, p. 1–16, 2018.

[41] R. Hennicker, A. Knapp, et H. Baumeister, « Semantics of OCL operation specifications », 2004.

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

[43] S. Vasilache, "Specification-based test case generation using dependency diagrams ," Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2016, WCECS 2016, 19-21 October, 2016, San Francisco, USA, pp 185–189.

Safaa Achour is currently a researcher at the Computer Science Laboratory. She received a Ph.D. degree in computer science in 2019 from the Faculty of sciences at the University of Ibn Tofail Kénitra, Morocco. Her research interests are in the areas of software testing; particularly, in model-based testing and Java Bytecode testing. (email: safaa.achour@uit.ac.ma).

Mohammed Benattou is a Full Professor of computer Science at the University of Ibn Tofail, Morroco. After completing his Ph.D at University of Blaise Pascal Clermont-Ferrand, he has held several positions in his French academic career: University of Pau, University of Orsay Paris XI, 3IL Institute of IT & Engineering of Limoges and Xlim Laboratory. His research areas include Software Engineering, and Computer Security and Reliability. (email: Mohammed.Benattou@uit.ac.ma).

Jean-Louis Lanet is a member of the Cidre research team at INRIA Rennes where he manages the LHS. He was a full Professor at the Computer Science Department of the University of Limoges (2007-2014). His researches focused on: Security of small systems like smart cards and software engineering. (e-mail: jean-louis.lanet@inria.fr<mailto:jean-louis.lanet@inria.fr>).