Formal Verification Technique for Consistency Checking between equals and hashCode methods in Java

Hiroaki Shimba†, Takafumi Ohta†, Hiroki Onoue†, Kozo Okano†and Shinji Kusumoto†

†Graduate School of Information Science and Technology, Osaka University
{h-shimba, t-ohta, h-onoue, okano, kusumoto}@ist.osaka-u.ac.jp

Abstract - Java objects used with the standard collection should override both of its equals and hashCode methods. Both of them need to satisfy the consistency rules, or unexpected behaviors may cause faults that are hard to detect. One of previous studies checks whether an equals method satisfies part of the consistency rule. In order to avoid the unexpected behaviors, however, it is necessary to check both equals and hashCode methods satisfy the rules. This research proposes a method which checks the consistency between equals and hashCode methods in Java. We model Java source code and check whether both methods satisfy the rules using an SMT solver called Z3. We have applied our proposed method to some projects in practice. As results, we have detected some of Java source code violating the rules.

Keywords: Java, equals method, hashCode method, Formal Verification, Satisfiability Modulo Theories(SMT)

1 Introduction

In Java, an equals method should be rightly overridden in a class, if its objects are compared. In order to guarantee an appropriate behavior of the collection framework, when a class overrides its equals method, its hashCode method also be overridden[1]. Therefore, Oracle API document defines some rules for the methods in an Object class[2]. For example, an equals method necessary to satisfy reflexive, symmetric and transitive properties. A method violating the rules may cause faults. It is well known that these faults are hard to detect [1][3][4]. Rupakheti et al. [5][6][7] presented a checker called EQ which is designed to automatically detect such an equals method violating the rules. EQ models an equals method and performs model checking to check whether the equals method satisfies part of the rules. Since EQ checks only equals methods, it cannot detect the class may cause fault when such an object is interacted with the collection framework. Also, EQ uses a model description language called Alloy which cannot model the bit operations. Hence, EQ cannot model equals methods using bit operations. Therefore, in order to avoid the unexpected behavior, we propose a new method which checks inconsistency between equals and hashCode methods. We use an SMT solver called Z3[8] to manipulate an arithmetic operations and bit operations which are often used in a hachCode methods. Since implementation patterns of equals and hashCode method are different, we propose new implementation patterns of hashCode methods. Also, we propose a method which converts Java code to an expression in a model description language called SMT-LIB[9]. We have applied our proposed method to some projects in practice. As results, we have detected some of Java source code violating the rules. The rest of this paper is organized as follows. Section 2, Section 3, Section 4, Section5, Section 6 and Section 7 present the consistency rules for equals and hashCode methods, a details of Z3, a motivation example, how convert Java code to SMT-LIB, an evaluation of our proposed method and discussion and a conclusion of this paper, respectively.

2 Consistent rules

This section presents the rules which equals and hashCode methods must satisfy.

2.1 Java Object class

Java Object class is defined as "root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class." by Oracle API document[2].

2.2 Consistent rules for equals methods

An equals method for Object class determines whether some other object supplied through its argument equals to this object. An equals method must satisfy the following four rules except a null object[2].

- reflexive: for any non-null reference value x, x.equals(x) should return true.
- symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- For any non-null reference value x, x.equals(null) should return false.

The equals method for Object class is defined as follows[2]. "The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object ($x == y$ has the value true). Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes."

```
public class Sample{
private int val;
private String str;
public boolean equals(Object obj){
if (obj == null)
return false;
if (this == obj)
return true;
if (!(obj instanceof Sample))
return false;
Sample that = (Sample) obj;
if (this.str == null){
return that.str == null;
}
return this.val == that.val && this.str.equals(that.str)
}
public int hashCode(){
return val + (this.str == null ? 0 : this.str.hashCode());
}
}
```

Figure 1: example of correct implementation of equals and hashCode methods

2.3 Consistent rules for hashCode methods

The hashCode method returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap. The hashCode method must satisfy the following two rules[2]. In this definition, information implies the returned value from the method invoked by its equals method or a field value used in the equals method. Thus, if there are some inconsistency between equals and hashCode methods, rule violation occurs.

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

The hashCode method for an Object class returns a different integer value for each different instances. Figure 1 shows an example of a correct implementation of equals and hashCode. The sample class has val and str as the integer and String type field values. The equals method for sample class determines whether an argument is the instance of the sample class after determines whether an object passed as the argument is identical to itself. Next, if the field value str is null, the equals method checks whether the str in passed object is also null. Finally, it determines whether the value of val and the string of str are identical. The hashCode method for the sample class concatenates the value of val and hash value of str. The sample class satisfies the consistent rules for both of equals and hashCode method.

3 Related works

Researches about implementation and design of method in Object class proposed the method that automatically generate the equals and hashCode methods. Rayside et al. have proposed a method which automatically generates the equals and hashCode methods which match the user demands by using a annotation of classes and methods [10]. This study performs dynamic analysis of source code. Grech et al. solved the problem of Rayside research that it consumes long time verifying cyclic objects by analyzing source codes statically[11]. Also, Jensen et al. proposed an annotation which guides the user when user copying objects by using clone method[12]. Recently, researches using model checking, SAT solver and SMT solver gain the attentions. Anastasakis et al. proposed a conversion method that converts class diagrams of UML with OCL to Alloy [13]. This research helps the developer who would like to perform verification about Alloy without knowledge of Alloy. Liu et al. suggested scalable bounded model checking by representing object oriented languages as bit vector of SMT solver[14]. This research supports high speed verification. Balasubramaniam proposed a constraint solver MINION that has high scalability and equips many functions[15]. Also, they proposed a method that automatically generates a constraint solver optimized to each domain[16]. This research helps generating the domain specific constraint solver. Burdy et al. proposed the method that statically verifies Java source code[17]. This method specifies the code location that may cause exceptions such as a NullPointerException. Also, it can verify Java source code annotated with JML. It is able to check whether each method satisfy the its constraints base on JML.

3.1 EQ

EQ checks whether equals method in Java satisfies the consistency rules. EQ receives a type hierarchy and outputs whether equals method satisfies the consistency rules. Here after, a type hierarchy is a structure of classes and interfaces represented as a DAG (Directed acyclic graph). Except Object class, classes and interfaces which have an inheritance relationship are belonged into the same type hierarchy. EQ consists of the following four steps. 1) Perform path analysis for equals method. 2) Analyze the pattern of equals method. 3) Convert Java code to a model described as Alloy. 4) Verify the model by alloy analyzer. EQ has two problems. One problem is that EQ dose not check whether a hashCode method satisfies the consistent rules. The other is that since alloy cannot model bit operators, alloy cannot model equals methods using bit operators. In this study, in order to solve those two problems, we use Z3 not Alloy.

```
public class COSString extends COSBase{
   public byte[] getBytes(){
      …
   }
   public boolean equals(Object obj){
   return (obj instanceof COSString)&&
      java.util.Arrays.equals
      (((COSString)obj).getBytes(),getBytes())
   }
   public int hashCode(){
      return getBytes().hashCode();
   }
}
```

Figure 2: hashCode methods violating consistency rules in PDFBox of Apache

3.2 Z3

SMT（Satisfiability Modulo Theories) problem is a decision problem for logical formulas expressed in first-order logic. An SMT solver solves SMT problems automatically. The SMT solver determines if a given logic formula which combination of theories expressed in first-order logic is satisfiable. If theories are satisfied, the SMT solver outputs assignments for variables that makes given theory satisfied. SAT problems described as theories that consists of only propositional variables. On the other hand, SMT problems described as theories that consist of propositional which can be many types such as Int similar to types in programming language. Also, SMT problems can define and use functions. In this study, we determine if both equals and hashCode method satisfy the consistency rules by using the SMT solver called Z3 exhaustively[3]. Z3 can use the arithmetic operations, bit vectors, arrays and recode types. Since an SMT solver searches the answer in bounded space exhaustively, it can verify there are no assignment which violates the consistency rules.

4 The Motivative example

In this section, we motivate this study by showing an example.

EQ[7] detected equals methods violating the consistency rules by experiments for four open source projects. The class implemented such equals methods may cause fault that is hard to detect. If an instance of a class which implements its equals method violating the consistency rules is used in the standard collection, unexpected behavior might cause faults. For example, if an instance of class which has the equals method violating reflexive is used in standard collection, a contains method of standard collection cannot determines correctly whether collection contains such a instance. Since in order to check equivalence of instances, a contains method of collection such a List uses equals methods, an unexpected behavior may occur. Also, if equals methods judge two instances are equivalent but these two instances return different hash values, hash-

```
public class ArEntry implements ArConstants{
   private String filename;
   public String getFilename() {
      return this.filename;
   }
   public boolean equals(Object it) {
      if (it == null || getClass() != it.getClass())
         return false;
   return equals((ArEntry) it);
   }
   public boolean equals(ArEntry it)
   if (this.filename == null)
      return (it.getfilename() == null);
   else
      return
         this.getFilename().equals(it.getFilename());
   }
   public int hashCode() {
      return super.hashCode();
   }
}
```

Figure 3: Conversion example of Java source code

Code methods cannot perform correct behavior. For example, HashMap may contain two instances judged equivalent by equals methods. Figure 2 shows our motivative example. This example shows an implementation of the hashCode method violating the consistency rules in PDFBox of Apache[18].

PDFBox uses java.util.Arrays.equals as equals method of COSString class. Also, PDFBox uses the hashCode method of byte array as the hashCode method of COSString class. Hence, equals method checks if two arrays have the same number of the elements and all corresponding pairs of the elements in the two arrays are equal The hashCode method checks these two arrays have the same memory address. Therefore, if instances of arrays are different and these array have the same elements with the same order, the equals method judges these two objects are equivalent but the hashCode method returns a different hash value for each other. In this case, HashMap may contain two instances judged equivalent by the equals methods. HashMap must not contain many instances judged equivalent by the equals methods. Since many cases are can be thought, it is difficult to detect the fault. For example, an insert procedure has fault and collection has fault.

In order to avoid such unexpected behavior, we propose new method that check whether both equals and hashCode methods satisfy the consistency rules.

5 Our proposed method

Our proposed method analyzes the Java code and models behavior of both of equals and hashCode methods in a model description language called SMT-LIB. The model is checked by Z3. Our proposed method receivesthe type hierarchy of the code and then outputs whether each of equals method sat-

```
;Class information
(declare-datatypes () ((Type ArEntry ArConstants UnderARC Object Null)) )
...
(declare-datatypes () (( Ref(Rfield (eqnum Int) (hsnum Int) (pointer Int)) )) )
(declare-datatypes () ((ArEntry(Arfield (filename Ref)) )) )
(declare-datatypes () (( Object(Ofield (ar ArEntry)(pointer Int)(class Type)))))
(declare-const this Object)
(declare-const that Object)
(declare-const other Object)
(declare-const nobj Object)
...
;method information
(define-fun equalsRef ((r1 Ref)(r2 Ref)) Bool
   (ite (and (and (not (= (pointer r1) 0)) (not (= (pointer r2) 0))) (= (eqnum r1)(eqnum r2))) true false ))
(define-fun equalsMain ((o1 Object)(o2 Object)) Bool
   (and (=> (or (= (class o1) ArConstants) (or (= (class o1) UnderARC)(= (class o1) Object)))
        (= (pointer o1)(pointer o2)))
        (=> (= (class o1) ArEntry) (and (and (not(= (pointer o2) 0)) (= (class o1)(class o2)))
        (or (and (= (pointer(filename (ar o1))) 0) (= (pointer(filename (aro2))) 0))
        (equalsRef (filename (ar o1)) (filename (ar o2)))) ) )
   )
)
(define-fun hashCode ((o1 Object)) Int(pointer o1))
;equality check
...
(assert (not (equalsMain this this) ) )
...
(assert (not(iff (equalsMain this that) (equalsMain that this)) ) )
...
(assert (not(=> (and (equalsMain this that) (equalsMain that other))
(equalsMain this other)) ) )
...
(assert (not(=> (not(= (pointer this) 0)) (not(equalsMain this nobj))) ) )
...
;hashCode check
(assert (not(=> (equalsMain this that) (= (hashCode this) (hashCode that) )) ) )
...
```

Figure 4: Conversion example of SMT-LIB

isfies the consistency rules. Our proposed method consists of the following four steps. 1) It perform path analysis for equals method. 2) It analyzes the pattern of the equals method. 3) It converts a given Java code to a model described in SMT-LIB. 4) It verifies the model by the Z3. Path analysis generates a control flow graph, and performs data flow analysis. Data flow analysis specifies what class is referred by a reference variable at each position of the source code and specifies what methods are called. Then, specified methods are inlinined into equals or hashCode methods if it is needed. Equals or hashCode methods perform some types of procedure. Therefore, pattern analysis classifies each method into some patterns. It is difficult to directly convert the hashCode procedures which contain loops including arithmetic operation or library calls, we analyze this procedure using heuristics operations. After pattern analysis, we convert Java code to SMT-LIB based on information from pattern analysis. Also, in order to check the violation of the obtained consistency rules, we also give some constraints to the SMT-LIB model. It is very difficult to model the first consistency rule of hashCode method. Please recall that the rule is “Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.” In order to model this rule, it is necessary to model the concept of the time. However, since first-order logic cannot represent the concept of the time, an SMT solver cannot check the first consistency rule of hashCode methods. Therefore, in order to resolve this problem, we introduce more strict consistency rule which replaces the first hashCode rule. On the other hand, since the second consistency rule of hashCode methods is representable in the first-order logic, an SMT solver can check the second consistency rule of hashCode methods directly. The substituted

consistency rule of hashCode method is as follows. We define the first rule below as the Subset rule and second one as the Equivalence rule.

- Subset rule: Set of the fields used in the hashCode methods must be subsumed by the set of fields used in equals methods.
- Equivalence rule: If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

Figure 3 and 4 show that an example of convert a Java source code (Fig.3) to a model written by SMT-LIB (Fig. 4). In this example, there are three classes in a type hierarchy. That is, an ArConstants interface, ArEntry class which implements ArConstants and overrides equals and hashCode method, and a class implementing ArConstants but do not override equals and hashCode method (this class represented as UnderARC in Fig.4). Figure.4 represents the SMT-LIB model of the source code in the type hierarchy. Figure.4 represents a declaration of types by the class information, a definition of the method behavior by the method information and the constraints used in validation by equality check.

5.1 Path analysis

Path analysis is similar to that of [7]. At first, our method searches equals and hashCode methods. Our method traces the inheritance relationship for a class which does not override its equals and hashCode methods. If we detect the class which overrides equals and hashCode methods, we regard equals and hashCode method of its parent class as the equals and hashCode method of such class. If there are no overrides of equals and hashCode methods in a inheritance relationship, we regard equals and hashCode method of Object class as the equals and hashCode in such class. Next, we analyze Java byte code using Soot[19] and generate its control flow graph. This control flow graph is represented by Jimple. Jimple represents a Java source code as three-address code , each expression consists of one operator, two operand, and one variable which stores the result of operation. Hereafter, we analyze a Jimple code generated by the Soot.

Next, our method performs path analysis. At first, our method enumerates paths using the obtained control flow graph. Next, our method performs data flow analysis for each path, and specifies what class is referred from a reference variable at each source code location and what methods are called. By this information, our method performs inlining the method invocations in equals or hashCode methods. However, since there are very large number of method invocation, our method limits the inlining. Our method only inlines the method invocations only in the type hierarchy. Also our method does not inline a getter method which is modeled as refer directly the field values. Although, Our method does not inlines outer methods, it models methods of Object class, wrapper classes, Array classes and Collections, because our behavior of these method are already well-known.

Finally, our method trims the path which is unreachable and not necessary to our model. Since our method models equals method as returns true, we trim the path which returns false. Also, in order to avoid modeling the null pointer exception, our method trims the path which includes uninitialized reference variables. Our method enhances the performance by trimming the path which is not necessary to model.

5.2 Analyzing the pattern of methods

In this step, our method analyzes the pattern of the procedure in equals and hashCode methods. By referring the modeling rules for each pattern, our method converts Java source code to SMT-LIB. Also, beside the pattern analysis, our method checks whether subset rule is violated in this step.

5.2.1 Analyzing patterns of equals methods

EQ introduce the six pattern of procedure in equals methods. Our method analyzes what pattern matches the equals methods. Six procedure pattern are equivalence checking of array, equivalence checking of List, equivalence checking of Set, equivalence checking of Map, type checking and state checking. Type checking checks whether there are type checking by instance operator in if expression, typecast by cast operator, type checking by getClass method in Object class. State checking checks whether there are equivalence checking of field values and checking reference variable is not null. Equivalence checking of array, List, Set, and Map checks whether there are comparison of the elements in each structure by loop.

5.2.2 Analyzing patterns of hashCode methods

We introduce the pattern of procedure of hashCode methods and defies the rules of each procedure. The hashCode methods procedure pattern are converting to int, bit operation and arithmetic operation in loop. Converting to int checks whether there are type converting by cast operation and type converting by library method of wrapper class. Arithmetic operation in loop checks whether there are procedure of add operation in loop.

5.2.3 Checking of the subset rule

Our method performs checking of subset rule. Our method collects a set of field variable used in equals and hashCode method by analyzing the equals method and hashCode method, and checks whether set of field variable used in hashCode methods are subsumed by set of field variable used in equals methods. If hashCode method invoke the method of parent classes and other methods since path analysis inlines the method of parent classes and other methods in hashCode methods, set of field variable used in hashCode method contains field variables used in such method. If values of variables in method of parent classes and other methods are changed, the change affects the return value of equals and hashCode methods. Therefore, since it is necessary to consider such field values, we substitute subset rule for the first rules of hashCode methods. Two cases occur in the consistence rule of hashCode methods. One is hashCode methods use fields values used in equals method In this case, if field values used in

```
(declare-datatypes () ((Type ArEntry ArConstants UnderARC
   Object Null)) )
(define-fun subof ((t1 Type) (t2 Type)) Bool
   (ite (or (= t1 Null) (= t2 Null)) false
      (ite (and (= t1 ArEntry) (= t2 ArConstants)) true
         (ite (and (= t1 UnderARC) (= t2 ArConstants)) true
            false
         )
      )
   )
)
(declare-fun instanceof (Type Type) Bool)
(assert (forall ((x Type) (y Type))
   (=> (subof x y) (instanceof x y))))
(assert (forall ((x Type) (y Type))
   (=> (and (instanceof x y) (instanceof y x))
      (= x y))))
(assert (forall ((x Type) (y Type) (z Type))
   (=> (and (instanceof x y) (instanceof y z))
      (instanceof x z))))
(assert (forall ((x Type)) (= (instanceof Null x) false) ))
(assert (forall ((x Type)) (=> (not(= x Null)) (instanceof x
   Object) )))
(assert (forall ((x Type)) (=> (not(= x Null)) (instanceof x x) )))
(assert (forall ((x Type)) (=> (not(= x ArEntry)) (not(instanceof x
   ArEntry)) )))
(assert (forall ((x Type)) (=> (not(= x UnderARC))
(not(instanceof x UnderARC)) )))
```

Figure 5: Model of the instanceof operation

equals method are not changed, hash values also not change. The other one is hashCode methods use not only field values used in equals method but also field values not used in equals methods. In this case, nevertheless field values used in equals method not change, hash values possibly change. In order to check this case, it is necessary to check relationships of field value used in equals and hashCode methods. Since it is necessary to check all method which modifies field values, analyzing it consumes much resource.

5.3 Conversion of Java source code to SMT-LIB

This step consists of the the following two steps. 1) basic structure conversion converts methods, inheritance relationships, classes and field values to SMT-LIB. 2) procedure of method conversion converts the procedure of the method to SMTLIB by using information obtained from the step of analyzing the pattern of methods.

5.3.1 Basic structure Conversion

Our method represents classes and fields by records in SMT-LIB. Our method defines fields used in equals and hashCode methods. It converts all primitive values to Ints in SMT-LIB. Since equals methods perform only comparison, Int has enough power to represent the result of equivalence checking.

Although hashCode methods perform any types of arithmetic operations, since hashCode methods usually perform typecast to int type before arithmetic operations, our method always converts primitive types used in hashCode methods to Ints. Our method converts the enumeration field to the enum type in SMT-LIB. Since reference variables of enum types possibly refer null, our method models add NULL value to the identifier introduced by the enum type. Also, since the enum type of hashCode methods invokes a hashCode method of Object class, our methods models the enum type of hashCode methods as returning the different values for each identifier. Our method defines reference type fields by introducing new record Ref representing a reference type. Ref represents the object that is out of type hierarchy. Our method models such an object based on the hypothesis that such a method satisfies the consistency rules of equals and hashCode methods. Ref defines a field variable that represents reference of its object. It is used in equivalence checking as Int type field. Our method defines the equals methods of Ref when an Ref object is used. Our method does not define hashCode methods of Ref. It models this as a reference of the hash values. Our method models the data structure of Java by arrays and lists. Our method represents arrays, Sets, Maps using arrays of SMT-LIB. An array of SMT-LIB is defined by specifying the type of its index and its type of elements. For example, specifying the type of index as Int represents the array. Set is also represented by adding a constraint in which elements are differ from each other to this array. Our method represents the inheritance relationship of a class by nest of records. However, it cannot model the behavior of instanceof which checks whether a class has a inheritance relationship between other classes. Hence, our method introduces type named Type which enumerates the type of adds null to all class in the type hierarchy. Our method models instanceof operator by representing the relation ship of Type. Figure 5 shows an example of an instanceof operation model. Definition of Object class defines all class as field. Object class represents the runtime objects and defines pointer as Int type. Type defines a field representing where the instances comes from.

Table 1: Part of simple μ conversion rules

$\mu(n_1+n_2)$	=	+ $\mu(n_1)$ $\mu(n_2)$
$\mu(n_1-n_2)$	=	- $\mu(n_1)$ $\mu(n_2)$
$\mu(n_1*n_2)$	=	* $\mu(n_1)$ $\mu(n_2)$
$\mu(n_1/n_2)$	=	/ $\mu(n_1)$ $\mu(n_2)$
$\mu(a_1==a_2)$	=	= $\mu(a_1)$ $\mu(a_2)$
$\mu(n_1<n_2)$	=	< $\mu(n_1)$ $\mu(n_2)$
$\mu(n_1>n_2)$	=	> $\mu(n_1)$ $\mu(n_2)$
$\mu(n_1>=n_2)$	=	>= $\mu(n_1)$ $\mu(n_2)$
$\mu(n_1<=n_2)$	=	<= $\mu(n_1)$ $\mu(n_2)$
$\mu(n_1!=n_2)$	=	not(= $\mu(n_1)$ $\mu(n_2)$)
$\mu(b_1\|\|b_2)$	=	or $\mu(b_1)$ $\mu(b_2)$
$\mu(b_1\&\&b_2)$	=	and $\mu(b_1)$ $\mu(b_2)$
$\mu(!b_1)$	=	not $\mu(b_1)$
$\mu(a_1 instanceof a_2)$	=	instanceof $\mu(a_1)$ $\mu(a_2)$
$\mu(a_1$. getClass())	=	class $\mu(a_1)$
$\mu(T_1$. class)	=	$\mu(T_1)$
$\mu(b_1?a_1{:}a_2)$	=	ite ($\mu(b_1)$) ($\mu(a_1)$) ($\mu(a_1)$)
$\mu(n_1\|n_2)$	=	bvor $\mu(n_1)$ $\mu(n_2)$
$\mu(n_1\&n_2)$	=	bvand $\mu(n_1)$ $\mu(n_2)$
$\mu(n_1\ \hat{}\ n_2)$	=	bvxor $\mu(n_1)$ $\mu(n_2)$

5.3.2 Conversion of the procedure of methods

Conversion of the procedure of method converts Java source code to SMT-LIB based on information obtained from step of analyzing the pattern of methods. First, our method generates expression trees for each expression represented as Jimple. Our method specifies the final expression returned by the return expression by tracing the expression tree and analyzing how valued of variables are calculated. The operation in expressions are converted by converting rules. Table 1 shows the simple converting rules of Java source code to SMT-LIB. The convert function converts Java source code to SMT-LIB, where bm and am represent an subexpression of boolean type and numerical type, respectively. Tm represents arbitrary types. Java represents an expression with infix notation while SMT-LIB represents expressions by prefix notation. Also, our method converts instanceof operator based on modeling previously described.

5.3.3 Conversion of equals methods

Our method converts equals methods based on six patterns obtained from the pattern analysis. Operations used in type checking are converted as shown in Table1. Since verification by an SMT solver is performed on the object level, cast operations used in equals method are not converted. Since statement checking compared values, the comparison expression is converted as Table 1. With regard to equivalence checking of arrays, Lists, Sets and Maps, our method models the method which performs comparison in the loop as performs comparison each element of an array. For example, lets' consider an instance of a class which has the array as the field, and performs equals method. Our method checks whether this equals method performs comparison of its field array with array of its argument by the same index. Next, our method checks whether a variable used in a loop header is used as index of array. If those two conditions are satisfied, our method determines it performs comparison. Most of loop operations in an equals match this pattern. Since other loop operations are rarely performed and SMT-LIB cannot evaluates statements dynamically, our method does not model such loop operations.

5.3.4 Conversion of hashCode methods

Our method converts hashCode methods based on the six patterns obtained from pattern analysis. Variables changed its type by cast operation or a method of Java class library is represented as Int type of SMT-LIB. Operands of bit operations are represented as 8bit type vector type,. Conversion results of the operations to Int types by applying the bv2int functions to the result. Although Int of Java is 32bit, if it models it as 32bit, modeling takes an enormous amount of time. Therefore, our method models it as an 8bit integer. Bit operations of hashCode methods operate two operands and not performs bit operations on specific one bit. Hence, our method can performs verification. Arithmetic operations in a loop are analyzed and our method determines what pattern matches the operations. Arithmetic operations in loop can be represented as expression, if the number of iteration is identical to the length of the array and arithmetic operations performed in loop do not contain nondeterministic values. However, the result of this operation is decided after the loop is terminated. Therefore our method limits the loop iteration. This is well used in bounded model checking. Our method calculates the result of the loop after 0 to 10 iterations. Our method cannot verify all cases but if our method decides a hashCode methods violate the rule, this decision is absolutely true. Similar reason of the equals method, our method does not model other loop operations.

```
unsat
  (error "line 74 column 17: model is not available")
unsat
  (error "line 80 column 22: model is not available")
unsat
  (error "line 86 column 28: model is not available")
unsat
  (error "line 92 column 22: model is not available")
sat
  ((this (Ofield (Arfield (Rfield 8 9 7)) 3 ArEntry))
  (that (Ofield (Arfield (Rfield 8 9 10)) 2 ArEntry)))
```

Figure 6: Results of verifying the code of Figure 4

5.3.5 Additional Constraints

Our method verifies the four consistency rules of equals methods and the equivalence rule of hashCode methods by an SMT solver. SMT solver solves the constraint and show assignments which is a set of values for the variables that satisfies all constraints. Therefore, in order to achieve an example of a type hierarchy which violates the consistency rule, our method introduces the negation of the consistency rules as the constraints.

5.4 Solving constraint by an SMT solver

Our method verifies the SMT-LIB expression which models Java source code using an SMT solver called Z3. In general, Z3 determines whether a given set of constraints is satisfiable or not. If it is unsatisfiable, it also outputs a counterexample which is a set of assignments of variables and interpretation of functions.

Since our method uses the negation of the consistency rules as the constraints in SMT-LIB, if Z3 outputs unsatisfiable, then we conclude that the source code does not violate the consistency rules. On the other hand, Z3 outputs satisfiable, we conclude that the source code violates the consistency rules In such a case, Z3 can output a set of assignments which makes the input true.

Figure 6 shows the results of verification by Z3 for the source code in Figure 4. The bottom line show the result of verifying the equivalence rule of the hashCode method and the other four lines are the results of verifying consistency rules of equals methods. Figure 6 shows that violation of the equivalence rule is detected. The optional outputs as as-

```
public class HCatFieldSchema implements Serializable {
   public enum Category {
      PRIMITIVE,ARRAY,MAP,STRUCT
   };
   String fieldName,typeString;
   Category category ;
   ...
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (obj == null)
         return false;
      if (!(obj instanceof HCatFieldSchema))
         return false;
      HCatFieldSchema other = (HCatFieldSchema) obj;
      if (category != other.category)
         return false;
      if (fieldName == null) {
         if (other.fieldName != null) {
            return false;
         }
      } else if (!fieldName.equals(other.fieldName)) {
          return false;
   }
      if (this.getTypeString() == null) {
         if (other.getTypeString() != null) {
             return false;
         }
      } else if (!this.getTypeString().equals(other.getTypeString())) {
         return false;
      }
         return true;
   }
   public int hashCode() {
      int result = 17;
      result = 31 * result + (category == null ? 0 : category.hashCode());
      result = 31 * result + (fieldName == null ? 0 : fieldName.hashCode());
      result = 31 * result + (getTypeString() == null ? 0 :
      getTypeString().hashCode());
      return result;
   }
}
```

Figure 7: A fixed HCatFieldSchema class

signments show that two ArEntry objects have the same field value but their references are differ.

6 Experiments

In this section, we evaluate our proposed method by experiment. We implement the verification function of the subset rule, a part of modeling to SMT-LIB and the verification function to our tool. We did not implement converting of bit operations and loops. These are one of the future work. Subsection 6.1 shows the results of applying our tool to some projects. The results show the effect of methods violating the subset rule. Subsection 6.2 shows the results that whether out tool can detect the violation of the consistency rules of equals methods. In the experiments, we firstly converted convert Java source code in practice to SMT-LIB manually. Then, we applied our tool to that model. Subsection 6.3 shows the execution time of our tool. Subsection 6.4 shows how often projects in practice violate the rules.

Table 2: Results of violation to the subset rule

Name	NumClass	Subset	Violation
Lucene	110	106	4

6.1 Evaluation of the subset rule

We applied our tool to Lucene4.6.0. Table 2 shows the results. Numclass represents the number of classes in which thier equals or hashCode methods are overridden. Subset represents the number of classes satisfying the subset rule. Violation represents the number of classes violating the subset rule.

We discuss about four classes which violate the subset rule. Two of four classes contain a field variable which stores the length of array and it used in only hashCode methods. The length of array can be calculated by the fields variable of array. Also, array is used in both equals and hashCode methods. Then, these classes are not completely violate the subset rule. Although these fields are declared with a keyword “final,” it guarantees that the reference variables refer always the same object, but it does not guarantee that the objects are not changed. Therefore, if the length of array changes, the field variable is not renewed and it does not store the correct value.

One of four class contains a field variable which stores the hash value already calculated for improvement of the performance. This class returns the hash value generated by converting memory address of object to integer value. Since this value does not change at runtime of application, the class does not completely violate the subset rule.

The last one class does not override its equals method, and invokes the equals method of Object class. Equals method of Object class does not use field values. However, this class overrides its hashCode method and it uses a field value. Therefore this class violates the subset rule.

6.2 Evaluation of the equivalence rule

We evaluated about the equivalence rule through the project in practice, HcatFieldSchema class of Apach Hive. This class receives a bug report which states that the class overrides its equals method but does not override its hashCode method at the past revision. This bug is fixed at the later revision. We manually modeled two revisions of this class. One contains the bug and the other fixed the bug. We can conclude our tool correctly work, if the following two conditions are satisfied. 1) Our tool detects that an unfixed class violates the consistency rules. 2) Our tool detects that a fixed class does not violate the consistency rules. Figure 6 shows that the source code of fixed class. This class does not have its parent class. An unfixed class does not override its hashCode method. If the hashCode method of the unfixed class is invoked, the unfixed class invokes the hashCode method of Object. The equals method of this class determines the equivalence of two objects by comparing field values. However, the hashCode method returns true if two objects are the same. Hence, this class violates the equivalence rule. Since the hashCode method of the fixed class returns a hash value by performing arithmetic operations about a field value used in the equals method, the fixed class does not violate the equivalence rule. We check

Table 3: Comparison of execution times

Name	Path length	Path analysis	Pattern procedure	analysis	Execution time
Lucene	16,970	12s	29s	1s	48s
Tomcat	257,590	38s	240s	2s	285s
JFreeChart	3,538,281	11,181s	11,491s	6s	22,689s

Table 4: The number of violated rules

Name	equals method				hashCode method		total
	reflexive	symmetric	transitive	null	subset	equivalence	
Lucene	2	0	0	0	4	1	7
Tomcat	11	3	4	3	14	7	35
JFreeChart	1	1	2	0	76	36	113

the violation of the equivalence rule by Z3. Z3 determines the unfixed class violates the equivalence rule, but the fixed class does not violates the equivalence rule. This result shows that our method can detect the implementation which violates the equivalence rule.

6.3 Execution times

In order to evaluate the cost of checking, we applied our tool to Lucene4.6.0, Tomcat8.0.1 and JFreeChart1.0.17. We compared the execution times. Figure 3 shows the results of this experiment. Path length, name of each step and time represent the total path length of each project, the execution time of each step and total execution time, respectively. Time represents the total execution time.

Theses result show that our proposed method is effective when checks it small or medium size projects. Our method can check large projects by limiting and reducing the search space. Execution time is approximately in proportion to the total pass length. We do not have an obvious answer to the cause of this result. Analyzing this cause is a future work. Also, analyzing procedure of a method and converting Java source code to SMT-LIB model consume over 50% of the total execution time. We can reduce the total execution time by improving the performance of these steps.

6.4 Evaluation of projects in practice

We evaluated how often projects in practice violate the consistency rules. We applied our tool to Lucene4.6.0, Tomcat8.0.1 and JFreeChart1.0.17.

Table 4 shows the results of this experiment. Each name of the rule column represents the number of implementations violating its rule.

We discuss about the cause of the violations of consistency rules. The causes of violating the rules of equals methods are those of [7]. That is, asymmetry null checking, invalid type checking at type hierarchy and miss typing. Also, we model the method invocations for fields as a nondeterministic function, and such modeling may generate wrong models. Three type hierarchies violating the rules caused by the wrong models. This problem can be solved by improving out tool. For example, we can solve this problem by using the information of method behavior from users for the method which is not inlined.

Regarding to the subset rule of hashCode methods, some classes contain a field variable which stores the hash value already calculated for improving the performance. This method returns the hash value generated by converting memory address of the object to an integer value. Since this value does not change at runtime of application, the class does not completely violate the subset rule. Also, regarding to the equivalence rule, there are many classes which override their equals methods but not override their hashCode methods, and violating this rule. This violation is only in JFreeChart and the other two projects do not contain such violation. Therefore, the policy of implementation of the project may affect this result. Consequently we claim that projects policy must contain the rule that if a class overrides the equals methods, then the class must override the hashCode methods. Also, there are two classes violate the equivalence rule of the hashCode methods. It is caused by their equals methods which violate the consistency rules.

7 Conclusion

In this paper, we proposed a method that verifies the consistency between both equals and hashCode methods. Also we have evaluated our method by experiments. Our method analyzes Java source code, and converts these code to SMT-LIB. Our method verifies whether the source code violates the consistency rules by using Z3. If they violate any of consistency rules, our method is able to output counter examples. Experimental results show that our method detects that some of real code includes a wrong method implementation which violates some of the consistency rules.

We will implement the functions which are not implemented our tool yet Also, we will evaluate the performance of our tool by applying our tool to many projects in practice. Experimental result shows that our method detects the inconsistency of some project, but does not shows that how many projects can be checked by our tool. We will apply our method to many projects and reveal it. These are future works.

Acknowledgments

This work is partially being conducted as Grant-in-Aid for Scientific Research S (25220003)

REFERENCES

[1] J. Bloch, “Effective Java,” Addison-Wesley, 2008.
[2] Oracle, “Java Platform, Standard Edition 7 API Specification,” 2013. http://docs.oracle.com/javase/7/docs/api/.
[3] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” ACM SIGPLAN Notices Homepage archive, pp.92-106, 2004.
[4] M. Vaziri, F. Tip, S. Fink, and J. Dolby, “Declarative Ob- ject Identity Using Relation Types,” Proceedings of the 21st European Conference on Object-Oriented Program- ming, pp.54-78, 2007.
[5] C.R. Rupakheti and D. Hou, “An Empirical Study of the Design and Implementation of Object Equality in Java,” Proceedings of the 2008 conference of the center for ad- vanced studies on collaborative research: meeting of minds, pp.111-125, 2008.

[6] C.R. Rupakheti and D. Hou, “An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java,” Proceedings of the 17th Working Conference on Re- verse Engineering, pp.205-214, 2010.

[7] C.R. Rupakheti and D. Hou, “Finding Errors from Reverse- Engineered Equality Models using a Constraint Solver,” Proceedings of the 28th IEEE International Conference on Software Maintenance, pp.77-86, 2012.

[8] L. deMoura and N. Bjorner, “Z3: An Efficient SMT Solver,” Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems, pp.337-340, 2008.

[9] Clark Barrett, Aaron Stump and Cesare Tinelli, “The SMT-LIB Standard Version 2.0,” 2010.

[10] D. Rayside, Z. Benjamin, R. Singh, J.P. Near, A. Milice-vic, and D. Jackson, “Equality and Hashing for (almost) Free: Generating Implementations from Abstraction Functions,” Proceedings of the 31st International Con-ference on Software Engineering,, pp.342-352, 2009.

[11] N. Grech, J. Rathke, and B. Fischer, “JEqualityGen: Gen- erating Equality and Hashing Methods,” Proceedings of the ninth international conference on Generative programming and component engineering, pp.177-186, 2010.

[12] T. Jensen, F. Kirchner, and D. Pichardie, “Secure the clones: Static enforcement of policies for secure object copy- ing,” Proceedings of the 20th European conference on Pro- gramming languages and systems: part of the joint European conferences on theory and practice of software, pp.317- 337, 2010.

[13] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, “UML2Alloy: A Challenging Model Transformation,” Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems, pp.436-450, 2007.

[14] T. Liu, M. Nagel, and M. Taghdiri, “Bounded Program Verification using an SMT Solver: A Case Study,” Proceedings of the 5th International Conference on Software Testing, Verification and Validation, pp.101-110, 2012.

[15] I.P. Gent, C. Jefferson, and I. Miguel, “Minion: A Fast, Scalable, Constraint Solver,” Proceedings of the 17th European Conference on Artificial Intelligence, pp.98-102, 2006.

[16] D. Balasubramaniam, C. Jefferson, L. Kotthoff, I. Miguel, and P. Nightingale, ”An Automated Approach to Generating Efficient Constraint Solvers,“ Proceedings of the 20129oiokpjg International Conference on Software Engineering, pp.661-671, 2012.

[17] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll, “An overview of JML tools and applications,” International Journal on Software Tools for Technology Transfer, pp.212-232, 2005.

[18] Apache, ”Apache PDFBox - A Java PDF Library,” 2012. http://pdfbox.apache.org/.

[19] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, “Soot a Java Optimization Framework,” Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, pp.125-135, 1999.