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

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

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

***Abstract*** - Java objects used with the standard collection should override both of its equals and hashCode methods. Both methods need to satisfy the consistency rules or unexpected behaviors may cause faults that are hard to detect. A previous study checked whether an equals method satisfies part of the consistency rule. To avoid unexpected behaviors, however, it is necessary to check that both the equals and the 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 applied our proposed method to some practical projects. As results, we detected some Java source code that violates 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. To guarantee the appropriate behavior of the collection framework, when a class overrides its equals method, its hashCode method should also be overridden [1]. Therefore, a document of Oracle API defines some rules for the methods in an Object class [2]. For example, an equals method is necessary to satisfy reflexive, symmetric, and transitive properties. A method violating the rules may cause faults. It is well known that such faults are hard to detect [1][3][4]. Rupakheti et al. [5]-[7] presented a checker called EQ, which is designed to automatically detect 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 a class that may cause a fault when its object interacts with the collection framework. Also, EQ uses a model description language called Alloy, which cannot model bit operations. Hence, EQ cannot model equals methods using bit operations. To avoid the unexpected behavior, we propose a new method which checks the inconsistency between equals and hashCode methods. We use a Satisfiability Modulo Theories (SMT) solver called Z3 [8] to manipulate arithmetic operations and bit operations which are often used in hashCode methods. Since the implementation patterns of equals and hashCode methods 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 applied our proposed method to some practical projects. As results, we detected some Java source code violating the rules. The rest of this paper is organized as follows. Section 2, Section 3, Section 4, Section 5, Section 6, and Section 7 present the consistency rules for equals and hashCode methods, a details of Z3, a motivating example, how to convert Java code to SMT-LIB, an evaluation of our proposed method and discussion, and the conclusion of this paper, respectively.

## 2 CONSISTENT RULES

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

### 2.1 Java Object Class

The Java Object class is defined as the "root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class." by an 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 this object. An equals method must satisfy the following four rules except for 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

```
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

is overridden to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes."

## 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 some inconsistency exists between equals and hashCode methods, a rule violation occurs.

- Whenever it is invoked on the same object more than once during the 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 does not need to 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 instance. 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, respectively. The equals method for the sample class determines whether an argument is the instance of the sample class after it 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 the hash value of str. The sample class satisfies the consistent rules for both the equals and the hashCode methods.

# 3 RELATED WORK

Research on the implementation and the design of a method in Object class proposed a method that automatically generates the equals and hashCode methods. Rayside et al. proposed a method which automatically generates the equals and hashCode methods which match the user demands by using an annotation of classes and methods [10]. This study performs a dynamic analysis of source code. Grech et al. solved the problem of the Rayside research, which consumes a long time to verify cyclic objects by analyzing source codes statically [11]. Also, Jensen et al. proposed an annotation which guides the user when the user copies objects by the clone method [12]. Recently, research using model checking by the Boolean Satisfiability Problem (SAT) solver and the SMT solver have gained attention. Anastasakis et al. proposed a conversion method that converts class diagrams of the Unified Modeling Language (UML) with the Object Constraint Language (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 a bit vector of the SMT solver [14]. This research supports high-speed verification. Balasubramaniam proposed the constraint solver MINION that has high scalability and provides many functions [15]. Also, they proposed a method that automatically generates a constraint solver optimized to each domain [16]. This research helps the generation of the domain-specific constraint solver. Burdy et al. proposed a 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 satisfies its constraints based on JML.

## 3.1 EQ

EQ [7] checks whether the equals method in Java satisfies the consistency rules. EQ receives a type hierarchy and outputs whether the equals method satisfies the consistency rules. Hereafter, a type hierarchy is a structure of classes and interfaces represented as a directed acyclic graph (DAG). Except Object class, the classes and interfaces which have an inheritance relation belong to the same type of hierarchy. EQ consists of the following four steps. 1) Perform path analysis for the equals method. 2) Analyze the pattern of the equals method. 3) Convert Java code to a model described as Alloy. 4) Verify the model by an Alloy analyzer. EQ has two problems. One problem is that EQ does not check whether a hashCode method satisfies the consistent rules. The other is that, since Alloy cannot model bit operators, Alloy cannot

```
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

model usual hashCode method using bit operators. In this study, to solve those two problems, we use Z3 not Alloy.

### 3.2 Z3

The SMT 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 is a combination of theories expressed in first-order logic, is satisfiable. If the theories are satisfied, the SMT solver outputs assignments for variables that make the given theory satisfied. SAT problems are described as theories that consist of only propositional variables. On the other hand, SMT problems are described as theories that consist of many propositional types, such as Int, which are similar to the types in programming language. Also, SMT problems can define and use functions. In this study, we determine whether both the equals and the hashCode methods satisfy the consistency rules by using the SMT solver called Z3 exhaustively [3]. Z3 can use arithmetic operations, bit vectors, arrays, and recode types. Since an SMT solver searches the answer in bounded space exhaustively, it can verify that no assignment violates the consistency rules.

## 4 EXAMPLE SHOWING MOTIVATION OF THIS STUDY

In this section, we present the motivation of 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 equals methods which may cause a 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 a class which has the equals method violating the reflexive rule is used in a standard collection, a contains method of the standard collection cannot determine correctly whether the collection contains such an instance. To check the equivalence of instances, a contains

```
int [] array1 = {3, 6, 2, 9};
int [] array2 = {3, 6, 2, 9};

COSString a = new COSString (array1);
COSString b = new COSString (array2);

// let assume that a.equals(b) but a.hashCode() != b.hashCode()

Set<COSString> database = new HashSet<COSString>();
database.add(a);
if (database.contains(b)) }
  // expect behavior
 System.out.println("Users want this.");
} else {
 // but this clause is executed
 System.out.println("However this statement is executed.");
}
```

Figure 3: Motivation Example

method of a collection such as List uses equals methods, an unexpected behavior might occur. Also, if equals methods judge two instances are equivalent but these two instances return different hash values, the hashCode methods cannot perform the correct behavior. For example, HashMap may contain two instances judged equivalent by the equals methods. Figure 2 shows an example of the motivation of this study. 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 the equals method of the COSString class. Also, PDFBox uses the hashCode method of a byte array as the hashCode method of the COSString class. Hence, the equals method checks whether 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 whether these two arrays have the same memory address. Therefore, if instances of the arrays are different and these arrays 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. In this case, HashSet may contain two instances judged equivalent by the equals methods.

It is important that both of equals method and hashCode method are rightly implemented, because incorrect implementation will causes unwanted behavior when programmer uses Java Collection Frame Work with it.

For example, let us consider the program in Fig. 3. Let assume that `a.equals(b)` holds but `a.hashCode() != b.hashCode()` also holds, in other words, we implement incorrectly hashCode method against its equals method. The program in Fig. 3 executes `else` clause which we do not expect.

Thus, it is important that we check whether both of equals method and hashCode method to be rightly implemented. The paper (EQ [7]) already has proposed a method for checking equals method, and therefore we focus on hashCode method.

To avoid such unexpected behavior, we propose a new method that checks whether both equals and hashCode methods satisfy the consistency rules.

Note that the implementation of both of equals method and hashCode method is programmers' obligation regardless the

```
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 4: Conversion example of Java source code

version of Java.

## 5 OUR PROPOSED METHOD

Our proposed method analyzes the Java code and models the behavior of both the equals and the hashCode methods in the model description language called SMT-LIB. The model is checked by Z3. Our proposed method receives the type hierarchy of the code and then outputs whether each equals method satisfies the consistency rules. The proposed method is based on static analysis.

Our proposed method consists of the following four steps. 1) It performs path analysis for the 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 Z3. The path analysis generates a control flow graph and performs data flow analysis. The 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 inlined into equals or hashCode methods if needed. The equals or hashCode methods perform some types of procedures. Therefore, pattern analysis classifies each method into some patterns. Because it is difficult to directly convert the hashCode procedures which contain loops including arithmetic operation or library calls, we analyze this procedure by using heuristic operations. After pattern analysis, we convert Java code to SMT-LIB based on information from the pattern analysis. Also, to check for violations of the obtained consistency rules, we give some constraints to the SMT-LIB model. It is very difficult to model the first consistency rule of the hashCode method. It should be recalled 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." To model this rule, it is necessary to model the concept of time. However, since first-order logic cannot represent the concept of time, an SMT solver cannot check the first consistency rule of the hashCode methods. Therefore, to resolve this problem, we introduce a more strict consistency rule which replaces the first hashCode rule. On the other hand, since the second consistency rule of the hashCode methods is representable in first-order logic, an SMT solver can check the second consistency rule of the hashCode methods directly. The substituted consistency rule of the hashCode method is as follows. We define the first rule below as the subset rule and the second one as the equivalence rule.

- Subset rule: Set of fields used in 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.

As Equivalence rule, Java specification gives a one-way rule. Therefore the rule (1)"`a.equals(b)` then `a.hashCode() == b.hashCode()`" is necessary while(1') "`a.hashCode() == b.hashCode()` impies `a.equals(b)`" does not need to be held. Our proposed method uses only (1) as Equivalence rule.

Figures 4 and 5 show examples of converting Java source code (Fig. 4) to a model written by SMT-LIB (Fig. 5). In this example, the type hierarchy has three classes. That is, the classes are an ArConstants interface, ArEntry class which implements ArConstants and overrides equals and hashCode methods, and a class implementing ArConstants that does not override equals and hashCode methods (this class is represented as UnderARC in Fig. 5). Figure 5 represents the SMT-LIB model of the source code in the type hierarchy. Figure 5 represents the declaration of types by the class information, a definition of the method behavior by the method information, and the constraints used for validation by an equality check.

### 5.1 Path Analysis

The path analysis is similar to that of [7]. First, our method searches equals and hashCode methods. Our method traces the inheritance relation for a class which does not override its equals and hashCode methods. If we detect the class which overrides the equals and hashCode methods, we regard the equals and hashCode methods of its parent class as the equals and hashCode methods of such a class. If no overrides of the equals and hashCode methods are found in an inheritance relation, we regard the equals and hashCode methods of Object class as the equals and hashCode in such a class. Next, we analyze Java byte code using Soot [19] and generate its control

```
;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 5: Conversion example of SMT-LIB

flow graph. This control flow graph is represented by Jimple. Jimple represents a Java source code as a three-address code, in which each expression consists of one operator, two operands, and one variable which stores the result of the operation. Hereafter, we analyze a Jimple code generated by Soot.

Our method performs a path analysis. First, our method enumerates paths by using the obtained control flow graph. Next, our method performs a 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. With this information, our method performs inlining of the method invocations in equals or hashCode methods. However, since the number of method invocations is very large, our method limits the inlining. Our method inlines the method invocations only in the type hierarchy. Also, our method does not inline a getter method, which is modeled as directly referring the field values. Although our method does not inline outer methods, it models methods of Object class, wrapper classes, Array classes, and Collections, because the behaviors of these methods are already well known.

Finally, our method trims the path which is unreachable and not necessary to our model. Since our method models the equals method as returning true, we trim the path which returns false. Also, to avoid modeling the null pointer exception, our method trims the path which includes uninitialized reference variables. In other words, our method enhances the performance by trimming the path not necessary to a 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 to the modeling rules for each pattern, our method converts Java source code to SMT-LIB. In addition to the pattern analysis,

our method checks whether a subset rule is violated in this step.

### 5.2.1 Analyzing Patterns of equals Methods

EQ introduce the six pattern of procedures in equals methods. Our method analyzes what pattern matches the equals methods. The six procedure patterns are equivalence checking of array, equivalence checking of List, equivalence checking of Set, equivalence checking of Map, type checking, and state checking. Type checking looks for the existence of the following: checking by an instance operator in an if expression, typecasting by a cast operator, type checking by getClass method in Object class. State checking looks for the existence of the equivalence checking of field values and checking a reference variable that is not null. Equivalence checking of Array, List, Set, and Map checks whether elements in each structure can be compared by a loop.

### 5.2.2 Analyzing Patterns of hashCode Methods

We introduce the pattern of the procedure of hashCode methods and define the rules of each procedure. The hashCode method procedure patterns are converting to int, a bit operation, and an arithmetic operation in loop. Converting to int checks for the existence of type converting by cast operation and type converting by library method of wrapper class. The arithmetic operation in loop checks the existence of the procedure of an add operation in loop.

### 5.2.3 Checking of the Subset Rule

Our method performs checking of the subset rule. Our method collects a set of field variables used in equals and hashCode methods by analyzing the equals method and the hashCode methods, and checks whether the set of field variables used in hashCode methods are subsumed by the set of field variables used in equals methods. If a hashCode method invokes the method of the parent classes and other methods, since the path analysis inlines the method of the parent classes and other methods in hashCode methods, the set of field variables used in the hashCode method contains field variables used in such method. If the values of variables in the 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 a subset rule for the first rules of hashCode methods. Two cases occur in the consistence rule of hashCode methods. One is that hashCode methods use fields values used in the equals method. In this case, if field values used in the equals method are not changed, hash values also do not change. The other case is that hashCode methods use not only field values used in the equals method but also field values not used in equals methods. In this case, nevertheless, the field values used in the equals method do not change, but hash values possibly change. To check this case, it is necessary to check relations of field values used in equals and hashCode methods. Since it is necessary to check all methods which modify field values, analyzing consumes many resources.

```
(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 6: Model of the instanceof operation

## 5.3 Conversion of Java Source Code to SMT-LIB

This step consists of the following two steps. 1) The basic structure conversion converts methods, inheritance relations, classes, and field values to SMT-LIB. 2) The procedure of the 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 comparisons, Int has enough power to represent the result of equivalence checking.

Since hashCode methods perform any type of arithmetic operations and 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 a 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 different values for each identifier. Our method defines reference type fields by introducing the new record Ref

Table 1: Some of the simple $\mu$ 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 $\mu(n_1)$ $\mu(n_2)$ |

representing a reference type. Ref represents the object that is out of the 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 the reference of its object. It is used in equivalence checking as the Int type field. Our method defines the equals methods of Ref when a 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, and 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 different from each other in this array. Our method represents the inheritance relation of a class by the nest of records. However, it cannot model the behavior of instanceof, which checks whether a class has an inheritance relation between other classes. Hence, our method introduces the type named Type which enumerates the type of adds null to all classes in the type hierarchy. Our method models the instanceof operator by representing the relation of Type. Figure 6 shows an example of an instanceof operation model. The definition of Object class defines all classes as a field. Object class represents the runtime objects and defines a pointer as an Int type. Type defines a field representing where the instance comes from.

### 5.3.2 Conversion of the Procedure of Methods

Conversion of the procedure of methods converts Java source code to SMT-LIB based on information obtained from the 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 the values of variables are calculated. The operation in expressions is converted by the 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 subexpressions of the boolean type and the numerical type, respectively. Tm represents arbitrary types. Java represents an expression with infix notation, whereas SMT-LIB represents expressions by prefix notation. Also, our method converts the instanceof operator based on modeling previously described.

### 5.3.3 Conversion of equals Methods

Our method converts equals methods based on the six patterns obtained from the pattern analysis. The operations used in type checking are converted as shown in Table 1. Since verification by an SMT solver is performed on the object level, cast operations used in the equals method are not converted. Since statement checking compares values, the comparison expression is converted as in Table 1. With regard to the equivalence checking of arrays, Lists, Sets, and Maps, our method models the method which performs a comparison in the loop as it performs a comparison of each element of an array. For example, let us consider an instance of a class which has the array as the field, and performs an equals method. Our method checks whether this equals method performs a comparison of its field array with the array of its argument by the same index. Next, our method checks whether a variable used in the loop header is used as the index of the array. If those two conditions are satisfied, our method determines it performs a comparison. Most loop operations in an equals method match this pattern. Since other loop operations are rarely performed and SMT-LIB cannot evaluate 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 the pattern analysis. A variable changed by its type by a cast operation or a method of the Java class library is represented as the Int type of SMT-LIB. Operands of bit operations are represented as 8-bit type vector types. Conversion results of the operations to Int types are obtained by applying bv2int functions to the result. Although Int of Java is 32 bits, if it models it as 32 bits, modeling takes an enormous amount of time. Therefore, our method models it as an 8-bit integer. Bit operations of hashCode methods operate two operands and do not performs bit operations on one specific bit. Hence, our method can perform verification. Arithmetic operations in a loop are analyzed and our method determines what pattern matches the operations. An arithmetic operation in a loop can be represented as expression, if the number of iterations 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

```
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 7: Results of verifying the code of Figure 5

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 method violates the rule, this decision is absolutely true. Similarly to the equals method, our method does not model other loop operations.

### 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. The SMT solver solves the constraint and shows the assignment, which is a set of values for the variables that satisfies all constraints. Therefore, to obtain an example of a type hierarchy which violates the consistency rule, our method introduces the negation of consistency rules as the constraints.

## 5.4 Solving Constraints by an SMT Solver

Our method verifies the SMT-LIB expression which models Java source code by using an SMT solver called Z3. In general, Z3 determines whether a given set of constraints is satisfiable. If it is unsatisfiable, Z3 also outputs a counter example, 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, if 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 7 shows the results of verification by Z3 for the source code in Fig. 5. The bottom line shows that the result of verifying the equivalence rule of the hashCode method and the other four lines are the results of verifying the consistency rules of equals method. Figure 7 shows that violation of the equivalence rule is detected. The optional outputs as assignments show that two ArEntry objects have the same field values but their references are different.

# 6 EXPERIMENTS

In this section, we evaluate our proposed method by experiment. We implement the verification function of the subset

```
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 8: A fixed HCatFieldSchema class

rule, part of the modeling to SMT-LIB and the verification function of our tool. We did not implement the converting of bit operations and loops. This is some of our 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 for whether our tool can detect violation of the consistency rules of equals methods. In the experiments, we first converted Java source code 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 the projects violated the rules.

## 6.1 Evaluation of the Subset Rule

We applied our tool to Lucene 4.6.0. Table 2 shows the results. Numclass represents the number of classes in which the 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 the four classes that violate the subset rule. Two of the four classes contain a field variable that stores the

Table 2: Results of violation of the subset rule

| Name | NumClass | Subset | Violation |
|---|---|---|---|
| Lucene | 110 | 106 | 4 |

length of the array and is used in only hashCode methods. The length of array can be calculated by the fields variable of the array. Also, array is used in both equals and hashCode methods. Therefore, these classes do not completely violate the subset rule. Although these fields are declared with a keyword "final", our method guarantees that the reference variables refer always to the same object, but it does not guarantee that the objects are not changed. Therefore, if the length of the array changes, the field variable is not renewed and it does not store the correct value.

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

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

## 6.2 Evaluation of the Equivalence Rule

We evaluated the equivalence rule through the HcatFieldSchema class of Apache Hive. This class receives a bug report which states that the class overrides its equals method but does not override its hashCode method in the past revision. This bug is fixed in the later revision. We manually modeled the two revisions of this class. One contains the bug and the other fixes the bug. We conclude that our tool works correctly, 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 the source code of the fixed class. This class does not have its parent class. The 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 involving a field value used in the equals method, the fixed class does not violate the equivalence rule. We check the violation of the equivalence rule by Z3. Z3 determines the unfixed class violates the equivalence rule, but the fixed class does not violate the equivalence rule. This result shows that our method can detect the implementation which violates the equivalence rule.

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: 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 |

## 6.3 Execution Times

To evaluate the cost of checking, we applied our tool to Lucene 4.6.0, Tomcat 8.0.1, and JFreeChart 1.0.17. We compared the execution times. Figure 3 shows the results of this experiment. The path length, the name of each step, and the time represent the total path length of each project, the execution time of each step, and the total execution time, respectively. Time represents the total execution time.

These results show that our proposed method is effective when it checks small or medium-sized projects. Our method can check large projects by limiting and reducing the search space. The 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 the cause is future work. Also, analyzing the procedure of a method and converting the 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

We evaluated how often the projects violate the consistency rules. We applied our tool to Lucene 4.6.0, Tomcat 8.0.1, and JFreeChart 1.0.17.

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

We discuss the causes of the violations of the consistency rules. The causes of violating the rules of the equals methods are those of [7]. That is, they are asymmetry null checking, invalid type checking at type hierarchy, and mistyping. 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 are caused by the wrong models. This problem can be solved by improving our tool. For example, we can solve this problem by using the information of method behavior from users for a method which is not inlined.

Regarding 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 the memory address of the object to an integer value. Since this value does not change at runtime of the application, the class does not completely violate the subset rule. Also, regarding the equivalence rule, many classes override their equals methods but

do not override their hashCode methods, and so they violate this rule. This violation is only in JFreeChart, not the other two projects. Therefore, the policy of implementation of the project may affect this result. Consequently, we claim that the projects policy must contain the rule that if a class overrides the equals methods, then the class must override the hashCode methods. Also, two classes violate the equivalence rule of the hashCode methods. This violation is caused by their equals methods that 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 evaluated our method by experiments. Our method analyzes Java source code and converts the code to SMT-LIB. By using Z3, our method verifies whether the source code violates the consistency rules. If thee code violates any of the consistency rules, our method is able to output counter examples. The experimental results show that our method detects that some of the real code includes incorrect method implementations which violate some of the consistency rules.

We will implement the functions which are not yet implemented in our tool. Also, we will evaluate the performance of our tool by applying our tool to many practical projects. Experimental results show that our method detects the inconsistency of some projects, but does not show how many projects can be checked by our tool. We will apply our method to many projects and examine the results. These are all future work.

## 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 Object Identity Using Relation Types," Proceedings of the21st European Conference on Object-Oriented Programming, 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 advanced 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 Reverse 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] C. Barrett, A. Stump and C. Tinelli, "The SMT-LIB Standard Version 2.0" (2010).
[10] D. Rayside, Z. Benjamin, R. Singh, J.P. Near, A. Milicevic, and D. Jackson, "Equality and Hashing for (almost) Free: Generating Implementations from Abstraction Functions," Proceedings of the 31st International Conference on Software Engineering, pp.342-352 (2009).
[11] N. Grech, J. Rathke, and B. Fischer, "JEqualityGen: Generating Equality and Hashing Methods," Proceedings of the 9th 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 copying," Proceedings of the 20th European conference on Programming 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 2012 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).

(Received November 20, 2014)
(Revised Feburary 23, 2015)

**Kozo Okano** received his BE, ME, and PhD degrees in Information and Computer Sciences from Osaka University in 1990, 1992, and 1995, respectively. From 2002 to 2015, he was an Associate Professor at the Graduate School of Information Science and Technology of Osaka University. In 2002 and 2003, he was a visiting researcher at the Department of Computer Science of the University of Kent in Canterbury, and a visiting lecturer at the School of Computer Science of the University of Birmingham, respectively. Since 2015, he has been an Associate Professor at Department of Computer Science and Engineering, Shinshu University. His current research interests include formal methods for software and information system design. He is a member of IEEE, IEICE, IPSJ.

**Hiroaki Shimba** received her BI and MI degrees from Osaka University in 2012 and 2014, respectively. His research interests include model driven software development, and consistency checking between equals and hashCode Methods in Java. He now works at Fuji Xerox Corp.

**Takafumi Ohta** received his BI and MI degrees from Tohoku University in 2013 and from Osaka University in 2015, respectively. His research interests include bug identification using concolic execution. He now works at NS Solutions Corporation.

**Hiroki Onoue** received her BI from Osaka University in 2014. His theme for the bachelor degree is “ implementation of consistency checking between equals and hashCode Methods in Java. ”He now works at Sharp Corp.

**Shinji Kusumoto** received his BE, ME, and DE degrees in Information and Computer Sciences from Osaka University in 1988, 1990, and 1993, respectively. He is currently a Professor at the Graduate School of Information Science and Technology of Osaka University. His research interests include software metrics and software quality assurance techniques. He is a member of the IEEE, the IEEE Computer Society, IPSJ, IEICE, and JFPUG.