IEICE TRANS. INF. & SYST., VOL.E102–D, NO.8 AUGUST 2019

PAPER

Consistency Checking between Java Equals and hashCode Methods Using Software Analysis Workbench

Kozo OKANO[†a)], ***Senior Member***, **Satoshi HARAUCHI**[††b)], **Toshifusa SEKIZAWA**[†††c)], **Shinpei OGATA**[†d)], ***Members***, ***and*** **Shin NAKAJIMA**[††††e)], ***Nonmember***

SUMMARY Java is one of important program language today. In Java, in order to build sound software, we have to carefully implement two fundamental methods `hashCode` and `equals`. This requirement, however, is not easy to follow in real software development. Some existing studies for ensuring the correctness of these two methods rely on static analysis, which are limited to loop-free programs. This paper proposes a new solution to this important problem, using software analysis workbench (SAW), an open source tool. The efficiency is evaluated through experiments. We also provide a useful situation where cost of regression testing is reduced when program refactoring is conducted.

key words: *software verification, Java, hash code, equivalence*

1. Introduction

Java is one of the most frequently used programming languages in constructing large and advanced software-intensive systems [1]. Their dependability relies on correct implementations of user-defined Java classes with respect to commonly agreed functional specifications. In particular, two of the basic methods, `hashCode` and `equals` are important methods for objects. Their implementations must satisfy design constraints that Java language specification defines [2]. The following is quoted from functional specification descriptions of class Object as defined in Java 8 API [3].

We can summarize these requirements as follows.

equals method:

1. **EQ reflexivity**: for any non-null reference value x, x.equals(x) should return true.
2. **EQ symmetry**: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
3. **EQ transitivity**: 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.
4. **EQ null**: For any non-null reference value x, x.equals(null) should return false.

both methods:

1. **EQHC consistency on object**: 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.
2. **EQHC consistency on time**: 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.

It is, however, not an easy task that programmers always observe those rules. There are many methods to resolve the problem. The related work and limitation is also shown in Sect. 2. This paper proposes a new approach for checking whether a user-defined class satisfies the rules. The proposed approach is based on the idea of [4] but utilizes the feature, such as equivalence checking on functions, of SAW (software analysis workbench) [5]. Though it has some limitations (see Sect. 4.4), the approach is also effective for cost reduction on regression testing.

We summarize the idea and main limitations here. We use SAW in order to resolve the problem automatically and more reliable way. This approach has a major limitation that the methods have fixed sized loop iterations due to SAW's capability. We also usually have to give a lemma to prove the equivalence in a reasonable time. However, this lemma is usually given easily.

The rest of this paper is organized as follows. Section 2 states our contribution and gives summary of related work. Section 3 is preliminaries. Section 4 describes the proposed method. Section 5 gives an application of our method for regression testing. Sections 6 and 7 show experimental results and discussion, respectively. Finally, Sect. 8 summarizes this paper.

Manuscript received July 14, 2018.
Manuscript revised January 26, 2019.
Manuscript publicized May 14, 2019.

[†]The authors are with Shinshu University, Nagano-shi, 380–8553 Japan.
[††]The author is with Mitsubushi Electric Corporation, Amagasaki-shi, 661–8661 Japan.
[†††]The author is with Nihon University, Koriyama-shi, 963–8641 Japan.
[††††]The author is with National Institute of Informatics, Tokyo, 101–8430 Japan.
a) E-mail: okano@cs.shinshu-u.ac.jp
b) E-mail: Harauchi.Satoshi@bc.MitsubishiElectric.co.jp
c) E-mail: sekizawa@cs.ce.nihon-u.ac.jp
d) E-mail: ogata@cs.shinshu-u.ac.jp
e) E-mail: nkjm@nii.ac.jp

DOI: 10.1587/transinf.2018EDP7254

2. Our Contribution and Related Work

2.1 Importance of the Approaches

In Java, `hashCode` method calculates a hash code of each object. The hash code is unique to the object, and is used, for example, for calculating locations in `HashMap`. Java also relies on `equals` methods for ensuring that given two objects are equivalent. Any two objects, considered equivalent, must have the same hash code value.

Listing 1 shows an actual implementation of the `hashCode` method violating the rules in an old revision of PDFBox of Apache [6]. PDFBox uses `java.util.Arrays.equals` as the `equals` method of the `COSString` class. The method `java.util.Arrays.equals` returns true if and only if the `COSString` arrays are the same size and have the same contents. This behavior is desired one. This code, however, implements `hashCode` method using array of Byte class. The array of Byte class inherits Object class, and `hashCode` method of Object class calculates a hash code differently as much as possible. Concretely the `hashCode` method returns the value based on the address of the object (the top address of array). Hence, the same contents of the `COSString` class will be stored at different locations in a HashSet because hash codes are different (see Fig. 1).

Listing 1: A `hashCode` method violating the rules in PDFBox of Apache

```
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();
  }
}
```

This wrong implementation was corrected at the latter revision of PDFBox of Apache. This story tells that satisfying the requirements is not easy for developing of real software.

In default, both `equals` and `hashCode` methods are inherited from its parent class or class `Object` when no such parent class is explicitly specified. Each user-defined class, however, sometimes requires to define its own these two methods because the methods must be so defined to faithfully follow its application semantics.

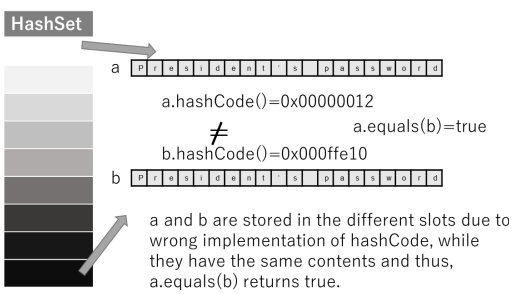

Fig. 1 Incorrect Hash Implementation

2.2 Related Work

Two major approaches exist. One of the approaches is automatic generation of the methods [7]–[9], while the other is verification of the methods. The first kind includes an approach using apache commons library. The library automatically generates an `equals` method which uses field variables programmers designate. The latter includes approaches presented in [10]–[12] and [4]. One of the advantages of these is that programmers can freely implement the methods.

Rupakheti *et al.* [10]–[12] have proposed a method for verification for only `equals` method based on alloy analyzer [13]. Alloy analyzer is a model finder and it uses first order logic with relation calculus. Okano *et al.* has proposed a method [4] which translates constraints and implementations of `equals` and `hashCode` methods into an SMT-lib2 [14] format, a common format for SAT/SMT solvers. Okano *et al.* has an advantage that can verify "**EQHC consistency on object** [4]." It, however, assumes that the method does not contain loop structures. The assumption is not applicable to most Java programs.

2.3 Contribution

Our contribution is that we propose a new, efficient verification based method. Our proposed method is based on our previous work [15]. The previous work, however, has some drawback in efficiency of checking. It also uses a complicated strategy to verify equivalence relation between the original method and its corresponding pure function. The verification is performed through their specifications, which does not always establish equivalence relation if the specifications are weak. Our new approach proposed in this paper overcomes this disadvantage under a moderate assumption. The efficient method proposed in this paper is the procedure **improvement** in Sect. 4. Furthermore, we apply the method to regression testing.

3. Preliminaries

3.1 Overview of SAW

Some recent formal verification techniques [16], [17] use JVM (Java Virtual Machine) and LLVM as their targets. An LLVM file is compiled from a C, C++, or Objective-C source file. LLVM is a virtual machine instruction set (Intermediate Representation) and usually used for code optimization in compilers. It, therefore, supports three-address code scheme and Static Single Assignment form, which facilitate static analysis for optimizing compiled code. LLVM has pointer types as well, which is mandatory for compliers

of C-family languages.

SAW (software analysis workbench) [5] is an open source software for analyzing C programs using LLVM. It needs an LLVM file as a part of its input. A user of SAW can analyze the LLVM using symbolic execution. The result of the execution is stored in AIG (And-Inverter Graphs) [18]. AIG data can be verified by a theorem prover called ABC [19]. ABC is especially good at equivalence checking [20] between two functions represented in AIG.

SAW, therefore, supports equivalence checking between two C functions given in the LLVM format. Both symbolic execution and equivalence checking functions are provided as commands of a script language used in SAW. SAW also supports property checking. SAW has been successfully applied to security domains such as Cryptographic Protocol Analysis.

SAW also uses SAT/SMT solvers [21]–[24] as its backend verification engines as well as ABC. Many SAT/SMT solvers have been made public [25] and are used as backend verification engines for other verification tools as well. The current version of SAW supports LLVM and JVM as well. It, however, does not support variable types such as references and arrays.

3.2 JML and OpenJML

JML [26] is an annotation language for Java using the notion of Design by Contract (DbC) [27]. ESC/Java2 [28] is a static analysis tool for JML. Recently, OpenJML [29] is expected to be a standard tool for JML. OpenJML supports simplify [30] which is used as a verification engine for ESC/Java2, and several verification engines such as Z3 and boogie [31] as well. Although JML is a useful notation, it cannot represent equivalence relation on the `equals` method [15]. It is because we must use multiple instances of `equals` explicitly in order to express reflexitivity and trasnsitivity, which is not allowed in the DbC style to use the keyword "result" without arguments.

As a consequence, we can say that the current version of Open JML tool is useful only for checking "**EQ null**" for our purpose.

4. Proposed Method

4.1 Basic Idea

We assume that both `equals` and `hashCode` methods are basically pure functions.

Pure functions produce no side effect on the environment. They do not modify their arguments nor field variables. Their return value solely depends on their arguments. Hence, if we invoke pure functions with the same arguments, we will always get the same return values. This assumption enables us to use SAW verification functions.

As a running example, we use a `hashCode` method with a loop structure in Listing 2.

Listing 2: example

```
public class EqualsHashCode {
  int x;
  int y;

  public boolean equals(EqualsHashCode obj) {
    if (obj == null)
      retun false;
    if (this.x == obj.x)
      return (this.y == obj.y);
    return false;
  }

  public int hashCode() {
    int r=17;
    int N=2;
    for (int i=0; i< N; i++) {
      r = 31*r+x;
    }
    r = 31*r+y;
    return r;
  }
}
```

The idea in our previous work [15] is that we prove that "x.equals(y) implies x.hashCode()= y.hashCode()," for any objects x and y using SAW.

There are several issues using SAW.

1. SAW does not fully support Java code, especially object reference variables.
2. Verification cost becomes high when the complexity of the code grows.

In order to resolve these issues, our previous work [15] decomposes the whole verification problem into a series of sub-tasks for proving equivalence verification on Java methods.

In this paper, we propose the following procedure, which uses an intermediate lemma and makes use of pure function presentation, first presenting a naïve approach (Sect. 4.2) and later a new approach for performance improvement (Sect. 4.3).

4.2 Naïve Approach

Here we propose our verification procedure.

First of all, like other approaches, we do not deal with "**EQHC consistency on time**" because it is a hard problem.

As described in Sect. 3.2, because we can verify "**EQ null**" with OpenJML, we use SAW for the others.

Recall the assumption we mentioned at the beginning of this section. We call the assumption **Pure function Assumption**.

[**Pure function Assumption**]

> Both `equals` and `hashCode` methods are basically pure functions.

Under **Pure function Assumption**, we can introduce a pure function `Pequals`() corresponding to the target `equals`() method.

Let us assume that the target class has field variables named, $x_0, x_1, \ldots, x_n$. Then `Pequals`() has its arguments as $x_0, x_1, \ldots, x_n$, $y_0, y_1, \ldots, y_n$, where x_i and y_i are corresponding to field variables of the receiver object and an argument object of the original `equals` method, respectively.

The body of Pequals() is almost the same as the original equals method.

Similarly we can introduce a pure function PhashCode() corresponding to the target hashCode() method. PhashCode() has $x_0, x_1, \ldots, x_n$ as its arguments.

Listing 3 is the rewritten version of Listing 2 by using pure function Pequals() and PhashCode().

Listing 3: example revised

```
public class EqualsHashCode {
  int x;
  int y;

  public boolean equals(EqualsHashCode obj) {
    if (obj == null)
      retun false;
    return Pequals(this.x, this.y, obj.x, obj.y);
  }

  public boolean Pequals(int x0, int y0, int x1, int y1) {
    if (x0 == x1)
      return (y0 == y1);
    return false;
  }

  public int hashCode() {
    return PhashCode(this.x, this.y);
  }

  public int PhashCode(int x, int y) {
    int r=17;
    int N=2;
    for (int i=0; i< N; i++) {
      r = 31*r+x;
    }
    r = 31*r+y;
    return r;
  }
}
```

In most cases, such transformation can be performed automatically. Hereafter, we assume that Listing 3 is given.

We perform verification on "**EQ reflexivity**," "**EQ symmetry**," "**EQ transitivity**," and "**EQHC consistency on object**," using the pure functions Pequals and PhashCode.

The verification procedure is summarized as follows.

1. We describe a property on "**EQ null**" in JML and verify it using OpenJML.
2. We describe pure functions corresponding to the target equals and hashCode methods, namely Pequals and PhashCode.
3. We assume that equals and hashCode can be implemented using Pequals and PhashCode.
4. We perform verification on "**EQ reflexivity**," "**EQ symmetry**," and "**EQ transitivity**" for Pequals using SAW (**thmE**).
5. Finally, we perform verification on "**EQHC consistency on object**" for Pequals and PhashCode using SAW (**thmH**).

Listing 4 shows an actual script.

Listing 4: verification script

```
jvm <- java_load_class "EqualsHashCode";

print "Extracting equals term";
x <- fresh_symbolic "x" {| [32] |};
y <- fresh_symbolic "y" {| [32] |};
z <- fresh_symbolic "z" {| [32] |};
w <- fresh_symbolic "w" {| [32] |};
t <- java_symexec jvm "Pequals" [("x", x), ("y", y),
   ("z", z), ("w", w)] ["return"] true ;
t' <- abstract_symbolic t;

x2 <- fresh_symbolic "xx" {| [32] |};
y2 <- fresh_symbolic "yy" {| [32] |};
print "Extracting hashCode term";
t2 <- java_symexec jvm "PhashCode"
   [("x", x2), ("y", y2)] ["return"] true ;
t2' <- abstract_symbolic t2;

print "Proving";

let thmE1 = {{ \z w  -> (t' z w z w)==1 }};
let thmE2 = {{ \z w u v -> ~((t' z w u v)==1)
  || ((t' u v z w)==1) }};
let thmE3 = {{  \z w u v i j-> ~(((t' z w u v)==1)
  && ((t' u v i j)==1)) || ((t' z w i j)==1) }};
let thmH = {{  \z w u v -> (~((t' z w u v)==1)
  || ((t2' z w) == (t2' u v)))}};
r <- prove_print abc thmE1;
print r;
r <- prove_print abc thmE2;
print r;
r <- prove_print abc thmE3;
print r;
r <- prove_print abc thmH;
print r;
print "Done.";
```

The first line of the script loads JVM. Then the script extracts a term representation for Pequals using the symbolic execution. The obtained term is substituted for t. The term t is finally translated into a function representation in SAW and substituted for t'.

Similarly a function representation of PhashCode is also placed in $t2'$.

The sentence `let thmE1 = {{ \z w  -> (t' z w z w)==1 }};` defines "EQ reflexivity" which follows definitions for the other theorems.

The command `r <- prove_print abc thmE1;` and others verify the theorems using ABC.

The scheme works for methods with small fixed constant iteration of loop structures, but when the constant becomes large, the verification time increases exponentially. We call this procedure **naïve-basic**. Because the method has a severe efficiency problems, we will consider improvement next.

4.3 Performance Improvement

A major factor for execution efficiencies is a verification task of **thmH**.

The theorem **thmH** includes term representations of two functions (Pequals and PhashCode), thus, the whole size of the term representation (**thmH**) becomes large. Moreover, as the number of loop iteration becomes large, the size also becomes large. In order to reduce the complexity, we introduce an intermediate lemma as in Fig. 2.

If we successfully verify the two theorems located at the upper side of Fig. 2, we can conclude that the goal

$$\frac{\text{Pequals}(\boldsymbol{x},\boldsymbol{y}) \rightarrow \text{LEMMA} \quad \text{LEMMA} \rightarrow \text{PhashCode}(\boldsymbol{x}) = \text{PhashCode}(\boldsymbol{y})}{\text{Pequals}(\boldsymbol{x},\boldsymbol{y}) \rightarrow \text{PhashCode}(\boldsymbol{x}) = \text{PhashCode}(\boldsymbol{y})}$$

Fig. 2 Using Verification Lemma

theorem located at the bottom also holds by Syllogism.

The lemma should be provided manually. In general, the lemma is given as follows.

For Pequals($\boldsymbol{x}, \boldsymbol{y}$),

$$\bigwedge_{i \in S} (x_i = y_i) \wedge (x_i \in \mathbb{Z}_{32} \wedge y_i \in \mathbb{Z}_{32}),$$

such that $\boldsymbol{x} = \{x_0, x_1, x_2, \ldots, x_n\}$, $\boldsymbol{y} = \{y_0, y_1, y_2, \ldots, y_n\}$, $S \subseteq \{0, 1, \ldots, n\}$, and $\mathbb{Z}_{32} = \{-2^{31}, 2^{31} - 1\}$.

$\mathbb{Z}_{32}$ is needed because JVM adopts 32-bit integers, while SAW assumes 64-bit integers. Such a restriction can be given by adding type [32] to the target variables in SAW.

The set S is also carefully determined because some of x_i (a field variable) is not used in `equals` for equivalence checking.

The whole procedure is the same as the the verification procedure descrived in Sect. 4.2 but 5th step is replaced by the sub-procedure described in this section.

We call this whole procedure **improvement**, which is an essential ingredient of the verification method proposed in this paper.

Note that if method `equals` does not support equivalence relation, then we can know it by failure of proof at the fourth step of the procedure **improvement**.

4.4 Limitation of the Proposed Method

Since the proposed method depends on SAW, its major limitation is also the same as the current version of SAW. It does not fully support reference variables, nor object-oriented specific features such as inheritance, polymorphism, and variety of types.

Though SAW can verify the equivalence fully automated, it uses an assumption that the loop is terminated in a fixed size and cannot deal with lists dynamically changing their sizes. In addition to the above, our approach assumes that both of the `equals` and `hashCode` methods are pure functions. Also our approach uses a lemma in order to reduce the cost of verification. The lemma should be given manually.

We have to notice, however, that:

1. from the theoretical point of view, the lemma is needed just for reducing the cost of verification, such as memory size and computation time. In other words, the lemma is not always needed.
2. the form of lemma is usually simple. Of course, for a case that equals method is implemented in a special way, we should give a suitable corresponding lemma, but it is usually easy to obtain from the code of the `equals` method.

In the classical approach in Bounded Model Checking, for loop, unwinding is performed, so that the fragment of body code is repeated k-times. Then, finite transitions starting from the initial state are converted to an SAT/SMT expression and verified. Various approaches to unbounded model checking have been studied in order to overcome the setting of the bound used in the bounded model checking approach, while preserving the advantages of the approach to convert to expressions in SAT/SMT. One approach is the k-induction method [32]. Several considerations, however, are required for dealing with the unreachable state. Another approach [33] is to use Craig's interpolation. This approach is basically an approximate solution. SeaHorn [34], a verifier for C programs uses ranking functions to check the termination of loops. It also abstracts a loop condition to a nondeterministic value.

KeY project [35] is used to prove several properties on Java program and has obtained many successes. Since it is basically a deductive approach, it needs several invariants and assertions generated manually. Although the strategy for the loop used in SAW version 0.2 is naïve compared with these approaches, since the iteration of the loop is limited to a fixed number of times, the verification is theoretically complete and sound.

Since SAW is is continuously developed, some of the limitations might be resolved in future.

5. Application to Regression Testing

An approach similar to the **improvement** procedure can reduce the cost of "a kind of" regression testing (verification). We demonstrate the usefulness through a concrete example.

Assume that we add a field variable z from Listing 2 and revise the `hashCode` method as shown in Listing 5. Such a process can be performed commonly when refactoring is applied to the class.

In such a situation, we want to ensure that the new `hashCode` method always returns the same value to be the value of the old `hashCode` method multiplied by 31. If such a fact is checked, we can use the new `hashCode` method instead of the old `hashCode` method with a little modification with divided by 31.

This checking is usually performed by an approach similar to regression testing. Regression testing, however, needs a lot of time and resources. Moreover, we cannot obtain firm confidence on the correctness of the results of the regression testing.

Applying our scheme, we can show the correctness of the revision by proving that the new `hashCode` always returns the same value as the value returned by the old `hashCode` value multiplied by 31 when $z = 0$.

Such verification can be performed using a verification script in Listing 6.

Listing 5: regression testing

```
public int hashCode() {
   int r=17;
   int N=2;
   for (int i=0; i< N; i++) {
     r = 31*r+x;
   }
   r = 31*r+y;
   r = 31*r+z;
   return r;
 }
```

Listing 6: verification script for regression testing

```
jvm <- java_load_class "EqualsHashCode";

x2 <- fresh_symbolic "xx" {| [32] |};
y2 <- fresh_symbolic "yy" {| [32] |};
print "Extracting hashCodeOld term";
t2 <- java_symexec jvm "PhashCodeOld"
  [("x", x2), ("y", y2)] ["return"] true ;
t2' <- abstract_symbolic t2;

x1 <- fresh_symbolic "x" {| [32] |};
y1 <- fresh_symbolic "y" {| [32] |};
z1 <- fresh_symbolic "z" {| [32] |};
print "Extracting hashCode term";
t1 <- java_symexec jvm "PhashCode"
  [("x", x1), ("y", y1), ("z", z1)] ["return"] true ;
t1' <- abstract_symbolic t1;

print "Proving";
let thm = {{ \x y -> (31*(t2' (x:[32]) (y:[32])
    == (t1' x y 0)) }};
prove_print abc thm;
print "Done.";
```

We call this procedure **regression**.

6. Experiments

We conducted several experiments of the verification tasks and measured their execution times. The setup of the experiments is summarized as follows.

- CPU Core i7 960 3.2 GHz
- MM 12 GB
- Windows 10 64-bit
- SAW version 0.2
- yices version 2.5.1
- cvc4 version 4.2.1-sjlj

The experiments use the following parameters.

- m: the number of field variables used in **equals** and **hashCode** methods; it varies from 2, 3, 4, to 5.
- N: the number of iteration in a loop of **hashCode** method; it varies from 2, 3, 4, 5, 10, to 20.
- back-end engines: ABC, yices, and cvc4.

6.1 Execution Times for Proving **thmE**

We measured execution times for proving **thmE** in Listing 2. Since parameter N does not affect these data, we only vary parameters m and back-end engines in Table 1. All of verification tasks return positively "pass."

6.2 Execution Times for Proving **thmH** in **naïve-basic**

Table 2 shows execution times for the verification on **thmH** in Listing 2, namely with **naïve-basic** method, Table 2, however, shows only cases where $m = 2$ and 5 as representatives.

6.3 Execution Times for Proving **thmH** in **improvement**

Tables 3 and 4 show results for verification based on **improvement**. We only show the times for **thmH**. Tables 3 and 4 correspond to proving theorem "Pequals(x,y) imply Lemma" (**thmH1** in short) and theorem "Lemma

Table 1 Execution time for **thmE** (sec.)

engine	$m = 2$	3	4	5
ABC	0.75	0.76	0.81	0.85
yices	0.75	0.80	0.85	0.89
cvc4	0.82	0.86	0.91	0.96

Table 2 Execution Time (sec.) for **thmH** in **naïve-basic**

engine	m	imp.	$N = 2$	3	4	5	10	20
ABC	2	c	19	3100	–	–	–	–
ABC	5	c	27	110	870	2132	–	–
yices	2	c	4	3120	–	–	–	–
yices	5	c	–	–	–	–	–	–
cvc4	2	c	0.74	0.77	0.75	0.77	0.77	0.78
cvc4	5	c	1.2	1.2	1.1	1.2	1.1	1.1
ABC	2	i	0.78	0.83	0.83	0.86	1.1	1.4
ABC	5	i	1.0	1.1	1.1	1.3	1.3	1.6
yices	2	i	0.75	0.76	0.77	0.75	0.77	0.81
yices	5	i	0.91	0.92	0.92	0.95	0.94	0.95
cvc4	2	i	0.76	0.82	0.81	0.84	0.84	0.83
cvc4	5	i	1.1	1.1	1.1	1.1	1.1	1.1

note: '–' stands for time over. Limitation is two hours.
Column "imp." means whether the target Java code is implemented correctly (c) or incorrectly (i).
Apart from time-out cases, verification returned **rightly** "invalid" and "pass" for i and c, respectively.

Table 3 Execution time for **thmH1** (sec.)

engine	imp.	$m = 2$	3	4	5
ABC	c	0.65	0.68	0.73	0.77
yices	c	0.63	0.71	0.75	0.79
cvc4	c	0.62	0.71	0.75	0.79
ABC	i	0.68	0.71	0.76	0.81
yices	i	0.68	0.73	0.77	0.81
cvc4	i	0.71	0.75	0.79	0.83

note: parameter N is not affect these data.
Column "imp." means whether the target Java code is implemented correctly (c) or incorrectly (i). All cases, verification returned **rightly** "invalid" and "pass" for i and c, respectively.

Table 4 Execution time for **thmH2** (sec.)

engine	m	$N = 2$	3	4	5	10	20
ABC	2	19	2710	–	–	–	–
ABC	5	27	63	540	1294	–	–
yices	2	0.62	0.62	0.66	0.63	0.63	0.64
yices	5	0.67	0.67	0.68	0.68	0.69	0.71
cvc4	2	0.62	0.64	0.64	0.64	0.63	0.65
cvc4	5	0.63	0.68	0.68	0.69	0.69	0.72

note: '–' stands for time over. Limitation is two hours.
For limitation of the space of the paper, we shows only for $m = 2$ and 5.
Apart from time-out cases, the verification returned positively, "pass."

Table 5 Execution time (sec.) for regression verification

engine	imp.	$N = 2$	3	4	5	10	20
ABC	c	0.71	0.71	0.71	0.71	0.71	0.74
yices	c	0.71	0.71	0.71	0.72	0.73	0.76
cvc4	c	0.72	0.73	0.74	0.74	0.74	0.76
ABC	i	0.77	0.83	0.83	0.86	1.1	1.5
yices	i	0.73	0.73	0.73	0.74	0.77	0.76
cvc4	i	0.74	0.76	0.75	0.76	0.78	0.81

note: Column "imp." means whether the target Java code is implemented correctly (c) or incorrectly (i). All cases, verification returned **rightly** "invalid" and "pass" for i and c, respectively.

imply PhashCode(x)=PhashCode(y)" (**thmH2** in short), respectively.

6.4 Execution Times for Proving **regression**

Table 5 shows execution times for regression testing (**regression**).

7. Discussion

Table 1 shows that all of the verification times for **thmE** are efficient. On the other hand, as we expected, Table 2 shows that verification tasks of **thmH** are time consuming and have tendency to increase exponentially. We can also find that incorrect implementations can be verified in short times regardless of engines.

Tables 3 and 4 are summaries for the case of checking "**thmH**" based on **improvement**. Table 4 shows that ABC cannot return any results for cases with large N, while both of yices and cvc4 show good performance for all cases. Though ABC is not improved drastically, but the values are improved against the **naïve-basic** method. From Tables 3 and 4, we can say that **improvement** scheme works well when we use SAT/SMT based engines.

The results of Table 5 imply that **regression** scheme is also useful in reducing the cost of regression testing for `hashCode` method in view of scalability.

Consequently, we can derive the following conclusions. The proposed method based on **improvement** is promising, though we have carefully to choose the verification engine. For regression testing, our proposed method **regression** is promising for all the engines.

8. Conclusion

We proposed a new and efficient method for verification on the consistency between `equals` and `hashCode` methods in Java. From the performance evaluation, we showed the usefulness of our proposed method.

Future work includes building a general tool for consistency checking which can deal with other verification engines.

Acknowledgments

The research is supported by the open collaborative research at National Institute of Informatics (NII), Japan (FY2016). The research is also being partially conducted as Grant-in-Aid for Scientific Research C (16K00094). Funding from Mitsubishi Electric Corp. and from Shinshu University are gratefully acknowledged. We also thank Mr. Shin Maruyama, a student of Shinshu University for helping SAW operation.

References

[1] N. Diakopoulos and S. Cass, "Interactive: The Top Programming Languages 2016," http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016 (accessed March 20th 2017).
[2] Oracle, "Java Platform, Standard Edition 7 API Specification," http://docs.oracle.com/javase/7/docs/api/ (accessed March 20th 2017).
[3] Oracle, "Java Platform Standard Ed. 8 AIP, class Object," https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html (accessed Oct. 20th 2017).
[4] K. Okano, H. Shimba, T. Ohta, H. Onoue, and S. Kusumoto, "Formal verification technique for consistency checking between equals and hashCode methods in Java," International Journal on Informatics Society, vol.7, no.2, pp.77–87, 2015.
[5] R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. Tomb, "Constructing semantic models of programs with the software analysis workbench," Proc. VSTTE 2016, 2016.
[6] Apache, "Apache PDFBox - A Java PDF Library," http://pdfbox.apache.org/ (accessed March 20th 2017).
[7] 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," Proc. 31st International Conference on Software Engineering, pp.342–352, 2009.
[8] N. Grech, J. Rathke, and B. Fischer, "JEqualityGen: Generating Equality and Hashing Methods," Proc. ninth international conference on Generative programming and component engineering, pp.177–186, 2010.
[9] T. Jensen, F. Kirchner, and D. Pichardie, "Secure the clones: Static enforcement of policies for secure object copying," Proc. 20th European conference on Programming languages and systems, pp.317–337, 2010.
[10] C.R. Rupakheti and D. Hou, "An Empirical Study of the Design and Implementation of Object Equality in Java," Proc. 2008 conference of the center for advanced studies on collaborative research: meeting of minds, pp.111–125, 2008.
[11] C.R. Rupakheti and D. Hou, "An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java," Proc. 17th Working Conference on Reverse Engineering, pp.205–214, 2010.
[12] C.R. Rupakheti and D. Hou, "Finding Errors from Reverse-Engineered Equality Models using a Constraint Solver," Proc. 28th IEEE International Conference on Software Maintenance, pp.77–86, 2012.
[13] D. Jackson, Software Abstractions, Revised Edition Logic, Language, and Analysis, MIT Press, 2011.
[14] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard Version 2.0," 2010.
[15] K. Okano, S. Harauchi, T. Sekizawa, S. Ogata, and S. Nakajima, "Equivalence checking of Java methods: Toward ensuring IoT dependability," Proc. 26th International Conference on Computer Communications and Networks, ICCCN 2017, pp.1–6, Aug. 2017.
[16] C.B. Lourenco, S.-M. Lamraoui, S. Nakajima, and J.S. Pinto, "Studying verification conditions for imperative programs," Proc. 15th International Workshop on Automated Verification of Critical Systems, AVoCS'15, 2015.
[17] S.-M. Lamraoui and S. Nakajima, "A Formula-based Approach for Automatic Fault Localization of Multi-fault Programs," Journal of Information Processing, vol.24, no.1, pp.88–98, 2016.
[18] J.A. Darringer, W.H. Joyner, Jr., C.L. Berman, and L. Trevillyan, "Logic synthesis through local transformations," IBM Journal of Research and Development, vol.25, no.4, pp.272–280, 1981.
[19] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," LNCS, vol.6174, pp.24–40, 2010.
[20] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol.21, no.12, pp.1377–1394, 2002.
[21] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," Proc. TACAS 2008, LNCS, vol.4963, pp.337–340, 2008.
[22] B. Dutertre, "Yices 2.2," Proc. CAV2014, LNCS, vol.8559,

pp.737–744, 2014.
[23] C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," Proc. 23rd international conference on Computer aided verification (CAV'11), pp.171–177, 2011.
[24] A. Cimatti, A. Griggio, B.J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," Proc. TACAS 2013, LNCS, vol.7795, pp.93–107, 2013.
[25] A. Biere, M. Heule, H. Van Maaren, and T. Walsh, Handbook of Satisfiability, IOS Press, 2009.
[26] 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.
[27] B. Meyer, Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997.
[28] P. Chalin, J.R. Kiniry, G.T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2," Proc. 4th International Symposium on Formal Methods for Components and Objects FMCO 2005, LNCS, vol.4111, pp.342–363, 2006.
[29] J. Sánchez and G.T. Leavens, "Static verification of ptolemyrely programs using openJML," Proc. 13th workshop on Foundations of aspect-oriented languages, pp.13–18, 2014.
[30] D. Detlefs, G. Nelson, and J.B. Saxe, "Simplify: A Theorem Prover for Program Checking," Journal of the ACM, vol.52, no.3, pp.365–473, 2005.
[31] M. Barnett, B.-Y.E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," Proc. 4th International Symposium on Formal Methods for Components and Objects FMCO 2005, LNCS, vol.4111, pp.364–387, 2006.
[32] B. Dutertre and L. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," Proc. 18th Conference on Computer Aided Verification (CAV 2006), LNCS, vol.4144, pp.81–94, 2006.
[33] K.L. McMillan, "Interpolation and SAT-Based Model Checking," Proc. 15th Conference on Computer Aided Verification (CAV 2003), LNCS, vol.2725, pp.1–13, 2003.
[34] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J.A. Navas, "The SeaHorn Verification Framework," CAV 2015, LNCS, vol.9206, pp.343–361, 2015.
[35] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich, Ed., Deductive Software Verification – The KeY Book – From Theory to Practice, Springer, 2016.

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 Associate Professor at Osaka University. In 2002 and 2003, he was a visiting researcher at Department of Computer Science of the University of Kent in Canterbury, and a visiting lecturer at School of Computer Science of the University of Birmingham, respectively. He is Associate Professor at Shinshu University. His current research interests include formal methods for software and information system design. He is a member of IEEE, IEICE, JSSST, and IPSJ.

Satoshi Harauchi received his BE and ME degrees in Information Sciences from Kyoto University in 1996 and 1998, respectively. Since 1998, he has been at the Advanced technology R&D center of Mitsubishi Electric Corporation and is currently interested in software engineering for social infrastructure system. He is a member of IEICE and JSASS.

Toshifusa Sekizawa received his MSc degree in physics from Gakushuin University in 1998, and Ph.D. in information science and technology from Osaka University in 2009. He previously worked at Nihon Unisys Ltd., Japan Science and Technology Agency, National Institute of Advanced Industrial Science and Technology, and Osaka Gakuin University. He is currently working at College of Engineering, Nihon University. His research interests include model checking and its applications.

Shinpei Ogata is Assistant Professor at Shinshu University. He received his BE, ME, and PhD degrees from Shibaura Institute of Technology in 2007, 2009, and 2012 respectively. His current research interests include model-driven engineering for information system development. He is a member of IEEE, ACM, IEICE, IPSJ, and JSSST.

Shin Nakajima is Professor at National Institute of Informatics, and is involved in research projects concerning to Formal Methods, Automated Verification, Software Testing, and Cyber-Physical Systems. He received his BSc, MSc, and Ph.D. degrees from the University of Tokyo, and is a member of IPSJ and JSSST.