社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

Javaにおける equals メソッドと hashCode メソッドの整合性の検査手法の提案

榛葉　浩章†　尾ノ上博樹†　岡野　浩三†　楠本　真二†

† 大阪大学大学院情報科学研究科
〒 565–0871 大阪府吹田市山田丘 1–5
E-mail: †{h-shimba,h-onoue,okano,kusumoto}@ist.osaka-u.ac.jp

あらまし　Java において，コレクションに格納されるオブジェクトは equals メソッドと hashCode メソッドをオーバーライドしている必要があり，これらのメソッドには満たすべき規則が存在する．この規則に違反したオブジェクトをコレクションに格納して使用した場合，正しい振る舞いをしなくなってしまい，さらにはそれによって引き起こされる欠陥を発見することが難しくなる．既存研究では equals メソッドのみを対象として満たすべき規則に違反しているかどうかを軽量形式手法で検査する手法が提案されている．しかし，equals メソッドをオーバーライドするときは常に hashCode メソッドもオーバーライドすべきであり，両方のメソッドの規則違反を検査すべきである．本研究では Java を対象として equals メソッドと hashCode メソッドの整合性を検査する手法を提案する．本手法では Java ソースコードをモデル化し，SMT ソルバ Z3 によって検査を行う．また，提案手法を実プロジェクトに適用し，実プロジェクトの中で規則に違反しているコードを発見することができた．

キーワード　Java，equals メソッド，hashCode メソッド，形式的検証，Satisfiability Modulo Theories(SMT)

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

Hiroaki SHIMBA†, Hiroki ONOUE†, Kozo OKANO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
Yamadaoka 1–5, Suita-Shi, Osaka, 565–0871 Japan
E-mail: †{h-shimba,h-onoue,okano,kusumoto}@ist.osaka-u.ac.jp

Abstract　Java classes must observe constraints on “hashCode” methods as well as “equals” methods, in order to behave correctly when their objects are interacted with the Java collection frameworks. One of researches on the consistency of such methods has proposed a lightweight formal method. This approach checks whether a given “equals” method observes the rules such as reflexivity, symmetry, and transitivity using Alloy Analyzer, a model finder. In general a class overridden with “equals” method must override its “hashCode” method properly. Therefore, we present a formal verification technique for consistency checking between “equals” and “hashCode” methods in Java. We translate Java code to SMT-LIB and verify it by Z3. Furthermore, our approach has been evaluated on open source projects. As a result of evaluation, we detected some violations in the projects.

Key words　Java, equals method, hashCode method, Formal Verification, Satisfiability Modulo Theories(SMT)

1. はじめに

Java において，オブジェクト同士の等価判定を行うクラスでは，そのクラスで求められている適切な等価性の定義に従って equals メソッドをオーバーライドする．また，equals メソッドをオーバーライドする場合には，コレクションフレームワークでの動作を適切に保証する等の理由により hashCode メソッドもオーバーライドする必要がある [1]．Java の Object クラスには，Oracle の API 仕様によって，これらのメソッドに対して満たさなければならない規則が規定されている [2]．例えば equals メソッドでは反射性，対称性，推移性などを満たしている必要がある．これらの規則に違反したクラスの実装は欠陥を

誘発するものであり，そのクラスのオブジェクトをコレクションに格納して使用すると，正しい振る舞いをせず，これにより誘発される欠陥を見つけることが難しくなる [1] [3] [4].

このような規則の違反をチェックする手法としてはRupakhetiらによる検査手法が存在する [5] [6] [7]．この手法では Java の equals メソッドを対象として，満たすべき規則に違反しているかどうかの検査を軽量的に行っている．しかし，この研究では equals メソッドのみの検査を行っており，equals メソッドがオーバーライドされるクラスにおいて，同様にオーバーライドされなければならない hashCode メソッドの検査は行っていない．また，既存研究で検査に用いているモデル記述言語にはビット演算が存在せず，正しくモデル化ができていないケースが存在する．これによりビット演算を用いた equals メソッドの検査は正しく行えていない.

そこで，本研究では Java を対象として equals メソッドと hashCode メソッドが満たすべき規則に違反しているかどうかを検査する手法を提案する．本手法ではハッシュコードの計算中で使用されることが多い算術演算やビット演算に対応するために SMT ソルバの Z3 を用いて検査を行う．また，equals メソッドと hashCode メソッド中で行われる処理のパターンは異なるため，変換する処理のパターンの提案と Java コードの SMT 制約式の記述フォーマットである SMT-LIB への変換方法の提案を新たに行う.

提案手法の評価として，提案手法をオープンソースプロジェクトに対して適用し，有用性の評価を行った．その結果，実プロジェクトの中で欠陥を誘発する実装を発見することができた.

以降，2 章では研究の背景となる諸技術について述べる．3 章では提案手法について詳しく述べる．4 章では提案手法の評価と考察を述べる．最後に 5 章で本報告のまとめを述べる.

2. 準　　備

本章では，本研究の背景知識や関連研究などについて簡単に述べる.

2.1 hashCode メソッド

hashCode メソッドは，オブジェクトのハッシュコードを返すメソッドである．hashCode メソッドが満たすべき規則は以下のように規定されている [2]．ここで情報とは，equals メソッド中で呼び出されたメソッドの戻り値や使用されているフィールドの値などを表す.

- equals メソッドで使用される情報が変更されなければ，hashCode メソッドは常に同じ整数を返す
- equals メソッドで等価と判定された 2 つのオブジェクトのハッシュコードは等しい

Object クラスの hashCode メソッドは異なるオブジェクトについては異なる整数値を返す．また，全てのクラスは Object クラスをスーパークラスとして持っているため，hashCode メソッドをオーバーライドしていないクラスには Object クラスの hashCode メソッドが継承される.

2.2 SMT ソルバ

SMT(Satisfiablility Module Theories) とは SAT [8] に算術式を追加したものである．SMT ソルバは変数の制約を満たす解を求める静的解析器である．SMT ソルバに問題とその制約を，変数や関数に関する制約式 (SMT 式) として入力すると，SMT ソルバは充足可能性を判定し，その解と充足する場合の変数への値の割り当てを出力する.

本研究では，検査対象の equals メソッドや hashCode メソッドが満たすべき規則に違反しているかどうかを網羅的に検査を行うために SMT ソルバの Z3 [9] を利用する．Z3 は算術演算やビットベクトル，配列，レコード型などを扱うことができる.

2.3 関 連 研 究

既存手法 [7] では Java を対象とした equals メソッドの検査を行っている．入力として 1 つの型階層を与え，型階層に含まれる equals メソッドが満たすべき規則に違反しているかどうかを出力する．ここで型階層とはクラスとインタフェースを DAG として表した構造である．Object クラスを除いた継承関係でつながっているクラスとインタフェースはすべて同じ型階層に含まれる．この手法では以下の手順で処理を行っている.

（1） equals メソッドに対してパス解析を行う
（2） equals メソッド中の処理のパターンを解析する
（3） Java コードを Alloy へ変換する
（4） Alloy Analyzer によって検査を行う

この既存手法には課題点が 2 つ存在する．1 つは equals メソッドをオーバーライドしているクラスがオーバーライドしなければならない hashCode メソッドの検査を行っていないことである．2 つ目は Alloy にはビット演算が存在しないために，ビット演算を使用する equals メソッドのモデル化が正しく行えておらず，正しい検査結果が得られないことである．本研究ではこの 2 つの問題を解決するために Alloy ではなく SMT ソルバの Z3 を用いて hashCode メソッドの検査を行う.

その他の関連研究について説明する．Object クラスのメソッドの設計や実装に関連する研究では，equals メソッドと hashCode メソッドを自動的に生成する手法がいくつか提案されている．Rayside らは equals メソッドや hashCode メソッドの計算に必要なクラスやフィールドにアノテーションを付加することで，ユーザーの目的に沿った equals メソッドと hashCode メソッドを自動的に生成する手法を提案している [10]．この手法ではソースコードの動的解析を行っている．Grech らはソースコードを静的解析することによって，Rayside らの手法の問題点である循環したオブジェクト図の検査に時間がかかることを改善した [11]．また，Jensen らは clone メソッドによってオブジェクトのコピーを行うときの方針を示すアノテーションを提案している [12].

3. 提 案 手 法

3.1 提案手法の概要

提案手法では Java コードの解析を行い，equals メソッドと hashCode メソッドの振る舞いを SMT-LIB 言語を用いてモデル化する．そして，SMT-LIB で記述されたモデルを SMT ソルバ Z3 により検査する．本手法の入力と出力は以下のとおりである.

```
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();
  }
}
```

図 1　変換例 (Java)

- 入力: 1 つの型階層
- 出力:規則に違反しているかどうか

また，本手法は以下の 4 つのステップから成り立っている．本報告では特に，パスの解析，メソッド中の処理のパターンの解析，SMT-LIB への変換について詳しく説明する．

- パスの解析
- メソッド中の処理のパターンの解析
- SMT-LIB への変換
- SMT ソルバによる検査

本手法では，hashCode メソッドが満たさなければならない 2 つの規則のうち，1 つ目は制約式で表すことが難しいため，より厳しい制約で代用して検査を行う．これは SMT ソルバでは時間の概念が存在しないため，モデル化することが難しいからである．2 つ目の規則は制約式で直接的に表すことができるため，SMT ソルバによって検査を行う．本手法で検査する hashCode メソッドの規則を以下に示す．以降で，1 つ目の規則をサブセット規則，2 つ目の規則を等価規則と呼ぶ．

- hashCode メソッドで利用されるフィールドの集合は equals メソッドで使用されるフィールドの集合のサブセットでなければならない (サブセット規則)
- equals メソッドで等価と判定された 2 つのオブジェクトのハッシュコードは等しい (等価規則)

変換例を図 1，2 に示す．この例では型階層に含まれているクラスは 3 つであり，インタフェースの ArConstants，ArConstants を実装していて equals メソッドと hashCode メソッドをオーバーライドしている ArEntry クラス，ArConstants を実装しているが equals メソッドと hashCode メソッドをオーバーライドしていないクラス (SMT-LIB では UnderARC となっている) である．この型階層に含まれるコードを SMT-LIB で表したものが図 2 で，上から順に，型に関する宣言，メソッドの振る舞いの定義，検査する制約の記述がされている．

3.2　パスの解析

パスの解析は基本的に既存手法 [7] を利用して行う．

まず，型階層中の各クラスの equals メソッドと hashCode メソッドを探索する．equals メソッドか hashCode メソッドのどちらかがオーバーライドされていないクラスは継承関係をたどっていき，オーバーライドされているクラスのメソッドをそのクラスの equals メソッドや hashCode メソッドとする．どこにもオーバーライドされていない場合は Object クラスのメソッドをそのクラスの equals メソッドや hashCode メソッドとして扱う．次に，Soot [13] を用いて Java バイトコードを解析し，検査対象メソッドの制御フローグラフを作成する．この制御フローグラフは Jimple で表現されている．Jimple は 3 番地コードでソースコードを表現したものであり，各文は 1 つの演算子と 2 つの被演算子，結果を格納する 1 つの変数の合計 4 つから構成されている．以降では Soot によって生成された Jimple 形式の中間コードを対象として解析を行っていく．

```
;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 (ar
o2))) 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)) ) ) )
...
```

図 2　変換例 (SMT-LIB)

次にパスの解析を行う．まず，メソッドの探索で作成された制御フローグラフを用いてパスの列挙を行う．次に，パスそれぞれについてデータフロー解析を行い，メソッド内の各位置での変数の値や参照変数がどのクラスのオブジェクトを指しているかを求める．この情報を利用してメソッド呼び出しの展開を行う．しかし，全てのメソッド呼び出しを展開すると多大なコストがかかってしまうため，本手法では展開するメソッドを型階層内に存在するメソッドに制限する．展開するメソッドのうち，ゲッターメソッドについては展開せずにフィールドを直接参照するようにモデル化を行う．検査対象の型階層に含まれないメソッド (外部メソッド) の呼び出しは基本的に展開を行わなず，非決定関数としてモデル化を行う．外部メソッドの呼び出しは基本的に展開を行わないが，Object クラス，ラッパークラス，Arrays クラス，コレクションのメソッドについてはあらかじめ振る舞いがわかっているため，それらのメソッドが呼び出されている場合は展開せずにモデル化を行う．

最後にパス解析の情報を用いて到達不能パスやモデル化に必

要ないパスの刈り込みを行う．equals メソッドのモデル化では true を返すパスの振る舞いをモデル化するため，false を返すパスを刈り込む．また参照変数に代入がされておらず null になっている可能性のあるオブジェクトが存在するパスの刈り込みを行う．このようにしてモデル化に必要ないパスを取り除き，解析のパフォーマンスを向上させる．

3.3 メソッド中の処理のパターンの解析

3.3.1 equals メソッドの処理のパターンの解析

equals メソッドの処理のパターンの解析では，既存手法で提示されている 6 種類のパターンがメソッド中に存在するかを解析する．解析するパターンは，型のチェック，状態のチェック，配列の等価判定，List の等価判定，Set の等価判定、Map の等価判定の 6 種類である．型のチェックでは if 文中で instanceof 演算子を使った型のチェック，キャスト演算を使った型のキャスト，Object クラスの getClass メソッドを使った型のチェックをしている演算が存在するかを解析する．状態のチェックではフィールドの値が等しいかの比較や参照変数が null でないのかチェックを行っているかを解析する．配列，List，Set，Map の等価判定ではループによってそれぞれのデータ構造の各要素の比較を行っているかを解析する．

3.3.2 hashCode メソッドの処理のパターンの解析

hashCode メソッドの処理のパターンの解析では，int 型への変換，ビット演算，ループによる算術演算が行われているかの解析を行う．int 型への変換ではキャストを用いた int 型への変換，ラッパークラスのライブラリメソッドを用いた int 型への変換が行われているかを解析する．ビット演算ではプリミティブ型に対してビット演算が行われているかを解析する．ループによる算術演算では，ループによってデータ構造の各要素を足し合わせる処理を行っているかの解析を行う．

3.4 SMT-LIB への変換

3.4.1 基本的な構造の変換

クラスとフィールドは SMT-LIB のレコード定義を用いて表現する．これによりクラスとそのクラスが保持するフィールドを定義することができる．プリミティブ型のフィールドは全て Int で表現する．これは，equals メソッドでは値の比較しか行わないため，等しいか等しくないかだけ判定できる型であればよいからである．hashCode メソッドでは算術演算を行っているが，キャストや変換メソッドを用いて int 型に変換をしてから算術演算を行っているため，Int 型で表しても問題がない．参照型のフィールドは，参照型を表す新しいレコード Ref を新たに定義し，それを用いて表現する．これは型階層外のクラスのオブジェクトを表しており，外部のクラスは hashCode メソッドと equals メソッドが満たすべき規則にしたがって実装されていると仮定してモデル化を行う．Ref にはそのオブジェクトのポインターを表すフィールド，オブジェクトの等価判定に使われるフィールド，ハッシュコードを表すフィールドの 3 つを Int 型のフィールドとして定義する．ポインターは 0 が null を表すものとしてモデル化を行う．

Java で用意されているデータ構造は SMT-LIB の配列やリストを用いて表現する．配列，Set，Map は SMT-LIB の配列を用いて表現する．SMT-LIB の配列はインデックスの型と要素の型を指定して定義を行う．インデックスの型として Int を指定すれば配列を表すことが可能である．また，この配列に対して，配列中の要素は互いに異なる値であるという制約を追加することで Set を表すことができる．キーと値を持つ Map については，インデックスの型に Int ではなく Map のキーの型を指定し，要素の型に Map の値の型を指定することで表現することができる．List は SMT-LIB のリストを用いて表現する．

表 1　単純な変換が可能な演算子に対する μ 変換の一部

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

クラスの継承関係はレコードの入れ子を用いて表現する．しかし，入れ子で表現しただけではあるクラスが他のクラスと継承関係になっているかを判定する演算子 instanceof の振る舞いをモデル化することができない．そこで，SMT-LIB の列挙型の定義を用いて Type という型名で検査対象の型階層中の全ての型に Null を追加した型を列挙する．そして，それらに成り立つ関係を SMT-LIB の関数定義を用いて記述することで instanceof 演算子のモデル化を行う．Object クラスの定義では型階層中のすべてのクラスをフィールドとして定義する．Object クラスは実行時のオブジェクトを表すものであり，ポインターを Int 型で定義し，そのオブジェクトがどのクラスのインスタンスであるかを表すフィールドを Type 型で定義する．

3.4.2 メソッド中で行われている処理の変換

メソッド中で行われている処理の変換では処理のパターンの解析で得られた情報を利用して SMT-LIB への変換を行う．まず，Jimple 形式で表現されたコードの各文に対して式木 (expression tree) を作成する．式の中で使用されている変数の値がどのように計算されるかを式木の情報を用いてたどっていき，return 文によって返される最終的な式を求める．そして，それらの式中の演算に対して変換ルールを適用する．Java コードから SMT-LIB への変換が単純に行える演算に対する変換ルールの一部を表 1 に示す．Java コードから SMT-LIB の変換関数を μ とし，型 boolean，数値型を持つ部分式をそれぞれ b_m，n_m で表す．任意の型を持つ部分式を a_m，任意の型を T_m で表

す．また，instanceof 演算については，前述の型階層の情報をもとにモデル化を行った instanceof 関数を呼び出すように変換を行っている．

3.4.3 equals メソッドの変換

equals メソッドの変換ではパターンの解析で得られた 6 つのパターンを SMT-LIB へ変換する．型のチェックで行われている演算は表 1 の通り変換を行う．equals メソッド中のキャスト演算に関しては変換を行わない．これは，SMT による検査は実行時のオブジェクトレベルでの検査を行うためである．状態のチェックでは，基本的に値の比較を行っているため，表 1 に従った比較文の変換を行う．配列，List，Set，Map の等価判定についてはループによってそれぞれのデータ構造に含まれる要素の比較を行っているもののみモデル化を行い，配列の各要素の値を比較するように SMT-LIB への変換を行う．equals メソッド中で行われているループ演算のほとんどがこのパターンに該当する．その他のループ演算に関しては，件数が少なく，SMT-LIB では式を動的に評価することができないため本手法では対応していない．

3.4.4 hashCode メソッドの変換

hashCode メソッドの変換ではパターンの解析で得られた 3 つのパターンを SMT-LIB へ変換する．キャストや Java クラスライブラリのメソッドを用いた int 型への変換がされている変数は SMT-LIB の Int 型で表現する．ビット演算の被演算子である変数は 8 ビットのビットベクトル型の変数で表し，演算の結果に対してビットベクトルを Int に変換する bv2int 関数を適用して Int に変換する．Java の Int は 32 ビットであるが，32 ビットでモデル化を行うと検査に膨大な時間がかかってしまう．そこで本手法では現実的な時間で解くことができる 8 ビットとする．hashCode 中のビット演算は 2 つの変数を対象とするビット演算を行っており，1 つの変数の特定のビットのみを操作する演算などは行っていないため，正しく検査することができる．ループによる算術演算では equals メソッドと同様に hashCode 中のループ演算が特定のパターンになっているかを調べる．配列の要素数と同じ回数だけループし，前回のループの結果に対して定数を掛けあわせて次の要素を足すという演算をしていれば式で表すことができる．しかし，この式はループ回数によって最終的な式が確定する．そこで本手法では，有界モデル検査で広く使用されている手法であるループ回数を決め打ちすることによってモデル化を行う．具体的にはループ回数を決め打ちして，０から１０までの場合を全て検査する．この方法では全ての場合を検査できるわけではないが，この方法で検査を行い，規則に違反していると判定された場合は確実に違反していると言える．また equals メソッドと同様の理由により，その他のループ演算には本手法では対応していない．

4. 評価と考察

4.1 評価の概要

本章では提案手法の評価について述べる．本研究ではサブセット規則の検査機能と，SMT-LIB へのモデル化および検査機能の一部を実装した．ビット演算とループの変換には対応できていない．評価 1 では，特定のプロジェクトに対して実装したツールを適用し，検出されたサブセット規則の違反例の考察を行う．評価 2 では，SMT-LIB へ変換して検査を行う等価規則に対する評価を行う．実プロジェクトのコードに対して手動で変換を行い，規則に違反した実装を検出できるかの評価を行う．評価 3 では実装したツールを複数のプロジェクトに適用し，実行時間の評価を行う．評価項目を以下に示す．

- 実プロジェクトの中で規則に違反している実装を検出できるか
- 実時間で検出することができるか

4.2 評価 1:サブセット規則について

表 2 サブセット規則の検査結果

プロジェクト名	対象クラス数	サブセット	サブセット違反
Lucene	110	106	4

4.2.1 結　　果

ツールを Lucene4.6.0 に対して適用した結果を表 2 に示す．対象クラス数は equals メソッドか hashCode メソッドのいずれかがオーバーライドされているクラスの数を表している．サブセットはサブセット規則を満たしているクラスの数，サブセット違反はサブセット規則に違反しているクラスの数を表している．

4.2.2 考　　察

サブセット規則に違反している 4 つのクラスに対して考察を行った．2 つのクラスはフィールドの配列の長さを別のフィールドで保持しており，hashCode メソッド中でのみ使用している．配列の長さは配列から求めることができ，配列自体は両方のメソッドで使用されているため，完全な違反であるとは言い切れない．これらのフィールドは final で宣言されているが，final 修飾子は参照変数が常に同じオブジェクトを指していることを保証するものであり，そのオブジェクトが変化しないことを保証するものではない．配列が変化してしまったときに配列の長さを保持しているフィールドの値を更新することができず，正しい値ではなくなってしまう．

他の 1 つのクラスは高速化のために一度計算したハッシュコードの値をフィールドで保持しており，hashCode 中でのみ使用している．このクラスはオブジェクトのメモリアドレスを整数に変換した値をハッシュコードとして返す．この値はアプリケーションの実行中では変化しないため，完全な違反であるとは言い切れない．

最後の 1 つのクラスでは equals メソッドがオーバーライドされておらず，Object クラスの equals メソッドが呼び出される．Obejct クラスの equals メソッドではフィールドの値は使用しない．hashCode メソッドはオーバーライドされており，フィールドの値を使って計算を行っているためサブセット規則の違反になっている．

4.3 評価 2:等価規則について

実験 2 では Apache Hive の HCatFieldSchema クラスを対象として評価を行った．このクラスは過去のリビジョンで，equals メソッドはオーバーライドしているが hashCode メソッドを

表 3　実行時間の比較

プロジェクト名	総パス長	前処理	パス解析	パターンの解析・変換	検査	実行時間
Lucene	16,970	5s	12s	29s	1s	48s
Tomcat	257,590	3s	38s	240s	2s	285s
JFreeChart	3,538,281	8s	11,181s	11,491s	6s	22,689s

オーバーライドしておらず，バグとして報告された．その後のリビジョンで修正され，hashCode メソッドをオーバーライドした．このクラスの修正前と修正後のコードに対して手動でモデル化を行った．このクラスには親クラスが存在せず，修正前のこのクラスのオブジェクトに対して hashCode メソッドを呼び出した場合，Object クラスの hashCode メソッドが呼び出される．equals メソッドではフィールドの値をもとにして等価判定を行っているが，hashCode メソッドではインスタンスが同じ場合にしか同じハッシュコードが返されず，等価規則に違反してしまう．修正後のコードでは hashCode メソッドがフィールドの値をもとにして計算を行っているため，等価規則に違反していない．修正前と修正後のコードを手動で変換したモデルに対して Z3 により検査を行った．修正前のコードでは等価規則の違反が検出された．また，修正後のコードでは違反は検出されず，equals メソッドと hashCode メソッドで整合性がとれていると判定された．この結果から，本手法により，実プロジェクトの中で規則に違反した実装を検出することが可能であることを確認できた．

4.4　評価 3:実行時間について

4.4.1　結　　果

ツールを Lucene4.6.0，Tomcat8.0.1，JFreeChart1.0.17 の 3 つのプロジェクトに対して適用し，実行時間の比較を行った．実行結果を図 3 に示す．総パス長は解析対象のパスの長さの合計を表している．各ステップ名は各ステップの処理時間を表している．また，実行時間は全体の実行時間を表している．

4.4.2　考　　察

3 つのプロジェクトの結果から，提案手法は小中規模なプロジェクトに対して有用であると言える．大規模なプロジェクトに対しては，プロジェクト中で解析対象を指定することによって適用が可能であると考えられる．実行時間に関しては，総パス長の増加に対して実行時間は大幅に増加している．この原因に関しては調査ができておらず，総パス長と実行時間の関係を明らかにすることが今後の課題である．また，メソッドの処理のパターンの解析・変換のステップがどのプロジェクトにおいても全体の実行時間の 50% 以上を占めており最長である．よって，このステップの改善を行うことで効果的な処理の高速化が可能である．

5.　あ と が き

本報告では欠陥を誘発する実装を発見することを目的として equals メソッドと hashCode メソッドの整合性の検査手法の提案，および実プロジェクトに対する評価を行った．本手法は Java コードを解析し，SMT-LIB への変換を行う．SMT-LIB に変換されたコードを SMT ソルバ Z3 により検査し，満たすべき規則に違反しているかどうかの解を求める．違反している場合は反例も同時に求めることができる．評価では，提案手法を実プロジェクトに適用し，いくつかの欠陥を誘発する実装を発見することができた．

今後の課題としては，実装できていない機能の実装や，様々なプロジェクトに対する評価実験があげられる．本研究では少数のプロジェクトに対する評価のみを行っているため，提案手法で提示したパターンに対する変換のみでどれほどのプロジェクトに対応できているのか明らかにできていない．多くのプロジェクトに提案手法を適用し，手法の有用性をより詳細に評価する必要がある．

謝辞　本研究の一部は科学研究費補助金基盤 (C)（21500036）と科学研究費補助基盤 (S)（25220003）の助成による．

文　　献

[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 the 21st 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] 梅村晃広，“SAT ソルバ・SMT ソルバの技術と応用,” コンピュータソフトウェア，vol.27，no.3，pp.24–35，2010.
[9] 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.
[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 ninth international conference on Generative programming and component engineering, pp.177–186, 2010.
[12] T. Jensen, F. Kirchner, and D. Pichardie, “Secure the clones: Static enforcement of policies for secure object 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] R. Vallée-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.