修士学位論文

題目

JavaにおけるequalsメソッドとhashCodeメソッドの整合性の検査

指導教員
楠本 真二 教授

報告者
榛葉　浩章

平成 26 年 2 月 5 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻

平成 25 年度 修士学位論文

Java における equals メソッドと hashCode メソッドの整合性の検査

榛葉　浩章

内容梗概

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 (反射性，対称性，推移性など) 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的手法によって検査する手法が提案されている．この手法では Java コードを Alloy に変換し，Alloy Analyzer によって検査を行っている．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，hashCode メソッドの満たすべき規則は equals メソッドに依存している．よって，hashCode メソッドは equals メソッドと整合性がとれていなければならず，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 を対象をして 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．既存研究では，Alloy Analyzer を用いて軽量的な検査を行っていたが，これは内部で SAT ソルバを使用しており，hashCode メソッドで行われる算術演算のモデル化には向いていない．そこで，本手法では Java コードを SMT ソルバへの入力言語である SMT-LIB に変換し，SMT ソルバによって検査を行う．SMT ソルバは SAT に算術式を追加したものであり，算術演算のモデル化が行いやすく，Alloy Analyzer では扱うことができないビット演算も扱うことができる．

また，提案手法を実プロジェクトに対して適用して評価を行った．結果として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．

主な用語

Java，Reverse Engineering，Satisfiability Modulo Theories(SMT)，Formal Verification

目 次

1 はじめに **1**

2 研究背景 **2**

2.1 Java の Object クラス 2

2.1.1 equals メソッド 2

2.1.2 hashCode メソッド 2

2.2 Soot 4

2.3 データフロー解析 4

2.4 Alloy 5

2.5 SMT ソルバ 5

2.6 Z3 6

2.7 既存手法 6

2.8 関連研究 8

3 研究動機 **10**

4 提案手法 **11**

4.1 提案手法の概要 11

4.2 パスの解析 12

4.2.1 検査対象メソッドの探索 13

4.2.2 パス解析 13

4.2.3 パスの刈り込み 16

4.3 equals メソッドと hashCode メソッド中の処理のパターンの解析 16

4.3.1 フィールドのサブセット判定 16

4.3.2 equals メソッドの処理のパターンの解析 17

4.3.3 hashCode メソッドの処理のパターンの解析 17

4.4 SMT-LIB への変換 17

4.4.1 基本的な構造の変換 18

4.4.2 メソッド中で行われている処理の変換 19

4.4.3 検査条件の挿入 23

4.5 SMT ソルバによる検査 23

5 評価と考察 **25**

5.1 評価の概要 25

5.2 評価 1:サブセット規則について 25

5.2.1 結果 25
5.2.2 考察 25
5.3 評価 2:等価規則について 26
6 あとがき **31**
謝辞 **32**
参考文献 **33**

図 目 次

1 equals メソッドと hashCode メソッドの実装例 3
2 SMT ソルバによる求解 6
3 型階層の例 7
4 既存手法の実装 8
5 hashCode メソッドの規則の違反例 11
6 変換例 (Java) 13
7 変換例 (SMT-LIB) 14
8 incetaneof 演算のモデル化 20
9 オブジェクトに成り立つ制約の記述例 21
10 Z3 による検査結果 24
11 修正前のコード 28
12 修正により追加された hashCode メソッド 29
13 変換された修正前のコード 29
14 変換された修正後のコード 30

表 目 次

1 検査対象のメソッド 15

2 展開せずにモデル化を行うメソッド 15

3 μ 変換：単純な変換が可能な演算 22

4 ツールの適用結果 25

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 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 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][5]．

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 [6] [7] [8] [9]．この手法では Java の equals メソッドを対象として，反射性，対称性，推移性などの性質に違反しているかどうかの検査を軽量的に行っている．手法の流れとしては，データーフロー解析によって到達不能パスやモデル化する必要のないパスの枝刈りを行った後に，equals メソッド中の処理のパターンの解析を行う．次に，解析した処理のパターンの情報を用いて Java コードを Alloy へモデル化する．最後に Alloy Analyzer によって，モデル化された Java コードが満たすべき規則に違反しているかどうかの検査を行う．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，Alloy にはビット演算が存在せず正しい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 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．hashCode メソッドの満たすべき規則は equals メソッドに依存しているため，equals メソッドの検査もあわせて行う．既存研究では Alloy Analyzer を用いて軽量的な検査を行っていたが，ハッシュコードの計算中で使用されることが多い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-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，提案手法を複数のオープンソースプロジェクトに対して適用し，提案手法の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 章では研究動機について説明する．4 章では提案手法について詳しく説明を行う．5 章では提案手法の評価と考察を述べる．最後に 6 章で本報告のまとめを述べる．

2　研究背景

本章では，本研究で用いる概念，ツールについて簡単に述べる．

2.1　Java の Object クラス

Java の Object クラスはクラス階層のルートであり，すべてのクラスはスーパークラスとして Object クラスを持っている．配列を含むすべてのオブジェクトは Object クラスのメソッドを実装する．よって equals メソッドや hashCode メソッドをオーバーライドしていないクラスのオブジェクトに対して，equals メソッドや hashCode メソッドを呼び出すと Object クラスのメソッドが呼び出される．

2.1.1　equals メソッド

Object クラスの equals メソッドは public boolean equals(Object obj) と定義されており，呼び出されたオブジェクトと引数で与えられるオブジェクトが等価であるかどうかを示すメソッドである．equals メソッドは null 以外のオブジェクトにおいて以下のような規則 (抜粋) を満たす必要がある．

- 反射性 (reflexive):null 以外の参照値 x について，x.equals(x) は true を返す
- 対称性 (symmetric):null 以外の参照値 x と y について，x.equals(y) は、y.equals(x) が true を返す場合だけ true を返す
- 推移性 (transitive):null 以外の参照値 x，y，z について、x.equals(y) が true を返し，かつ y.equals(z) が true を返す場合に，x.equals(z) は true を返す
- null でない任意の参照値 x について，x.equals(null) は false を返す

Object クラスの equals メソッドはもっとも比較しやすいオブジェクトの同値関係を実装している．null 以外の参照値 x と y について２つのオブジェクト x と y が同じオブジェクトを参照する (x == y が true) 場合にだけ true を返す実装となっている．この実装は満たさなければならない規則をすべて満たしている．また，通常 equals メソッドをオーバーライドする場合は，hashCode メソッドを常にオーバーライドして，等価なオブジェクトは等価なハッシュコードを保持する必要があるという hashCode メソッドの規則に従う必要がある．

2.1.2　hashCode メソッド

Object クラスの hashCode メソッドは public int hashCode() と定義されており，オブジェクトのハッシュコードを返すメソッドである．このメソッドは java.util.Hashtable によって提供されるようなハッシュテーブルで使用するために用意されている．hashCode メソッドが満たすべき規則 (抜粋) は以下のように規定されている．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.

```java
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());
 }
}
```

図 1: equals メソッドと hashCode メソッドの実装例

- Java アプリケーションの実行中に同じオブジェクト上で複数回呼び出される場合は必ず，このオブジェクトに対する equals による比較で使われた情報が変更されていなければ，hashCode メソッドは同じ整数を一貫して返さなければならない．
- equals(Object) メソッドで 2 つのオブジェクトが等価とされた場合，どちらのオブジェクトで hashCode メソッドを呼び出しても結果は同じ整数値にならなければならない

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．これは，オブジェクトの内部アドレスを整数値に変換する形で実装されている．この実装は hashCode メソッドが満たすべき規則をすべて満たしているが，これらの規則は equals メソッドの実装に依存することに留意しなければならない．

また，equals メソッドと hashCode メソッドの実装例を図 1 に示す．このクラスは int 型のフィールド val と String 型のフィールド str を持っている．equals メソッドでは，引数で与えられたオブジェクトが自分と等しいかのチェックを行った後，Sample クラスのインスタンスであるかのチェックを行っている．次に自分の str が null の場合は，相手の str も null かのチェックを行い，最後に val の値が等しいかのチェックと str が指している文字列が等しいかのチェックを行っている．hashCode メソッドでは val の値と str のハッシュコードを足し合わせている．この実装は equals メソッドと hashCode メソッドの満たすべき規則に従った実装である．

2.2 Soot

Soot[10] は McGill 大学のプロジェクトとして開発されている Java バイトコードの解析を行うフレームワークであり，Java バイトコードを解析して 4 種類の中間コードを生成することができる．中間コードの種類としては以下の 4 種類が存在する．また Soot は中間コードを生成するときに制御フローグラフ (CFG) も自動的に生成する。

- Baf : もっともバイトコードに近い表現
- Jimple:3 番地コードによる表現
- Shimple:SSA(静的単一代入) 形式による表現
- Grimp:逆コンパイルやコード解析に適した表現

Soot の特徴は処理が複数のフェイズに分割されており，任意のフェイズに新たな処理を追加できることである．Soot に Java バイトコートを解析させるだけでも中間表現を取り出すことは可能だが，フェイズにコード変換や最適化処理を挿入することで様々な出力を得ることができる．

本研究では Java バイトコードを 3 番地コードで表現した Jimple を利用する．3 番地コードはコンパイラにおける中間言語の一種であり，各命令を 2 つの入力と 1 つの出力のアドレスを指定する形で表現する．3 番地コードの各命令は命令コード、オペランド 1、オペランド 2，結果の 4 つで表現され，各命令が正確に 1 つの基本演算を実装している．また、SSA は 3 番地コードを改良したものである．

2.3 データフロー解析

データフロー解析はプログラムの様々な位置で実行時の情報を得るための手法である．データフロー解析では制御フローグラフ (CFG) を用いて変数の値が伝播するかどうかなどの情報を集め，変数などが取りうる値の集合に関する情報を収集する．データフロー解析によって到達不能コードや初期化していないローカル変数，null オブジェクトの参照などを検出することができる．データフロー解析はプログラムの妥当性の検査やデバッグ，保守，テストなどに利用されている．

データフロー解析では制御フローグラフの各ノードについてデータフロー方程式を設定し，全体として安定した状態になるまでそれらの式を繰り返し計算していく．本研究では path-sensitive なデータフロー解析を行う．path-sensitive とは条件分岐命令において条件判断にかかわる情報の解析を行う．例えば，条件分岐の条件が x > 0 の場合，そのまま分岐しないで処理を続行する場合は x <= 0 として解析を行い，分岐した場合は x > 0 として解析を行う．このようにしてノードの情報だけでなく条件分岐の情報の解析も行う．

2.4 Alloy

Alloy[11] は集合と関係からなる一回述語論理を用いて規則を記述する言語で Z 言語 [12] をもとにしている．Alloy ではシグネチャと述語を用いてモデルを記述する．

Alloy で記述した仕様は Alloy Analyzer[13] によって検査することができ，仕様を満たす例 (インスタンス) を有界網羅的に探索する．このとき実行コードやテストケースなどは必要ない．解析の際には，スコープにより解析範囲を制限することで，特定の範囲内を網羅的に解析できる．スコープはモデルの中で定義された各インスタンスの上限数を表している．また，Alloy Analyzer では検査結果を図で視覚的に示してくれるので直感的に理解しやすい．

Alloy の内部では SAT ソルバ [14] を利用しており，用意された複数の SAT ソルバの中からどれを用いて検査を行うかを指定できる．Alloy Analyzer は Alloy で記述されたモデルを充足可能性問題 (SAT) に変換して SAT ソルバに渡し，検査結果を受け取って視覚的に出力する役目を果たしている．

2.5 SMT ソルバ

SMT とは，Satisfiability Modulo Theories の略で SAT[14] 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．そして，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．

SAT では true か false のどちらかの値をとる命題変数 (Bool 変数) のみで問題を記述しなければならないが，SMT ではプログラムで使用するような Int 型や Real 型の変数を用いて問題を記述することができる．また，述語 (関数) を定義して使用することもできる．一般に，SMT 式は SAT 式で記述するよりも記述量が減り，人間に理解しやすいという利点がある．

図 2 は以下の方程式の解を SMT ソルバで求める例である．declare-const コマンドで割り当てを求める変数の宣言を行い，assert コマンドでそれらの変数が満たさなければならない制約を記述している．

$$\begin{cases} x + y = 10 \\ x + 2y = 20 \end{cases}$$

SMT ソルバには様々な実装があり [15][16][17][18][19]，それぞれに扱える論理が異なる．また，SMT の性能向上を目的として SMT ソルバの性能を競い合う競技会 [20] が開催されている．ここではいくつかの問題のクラスに対して実行時間や消費メモリなどを競い合う．競技結果は公開されており，この情報を利用して目的に沿った SMT ソルバを選択することができる．また，SMT ソルバへの入力は SMT-LIB[21] 形式で行う．SMT-LIB は SMT ソルバの競技会が定めた標準の入力形式であり，この入力形式に対応することが事実上の標準となっている．

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網羅的に検査を行うために SMT ソルバの Z3[19] を利用する．SMT ソルバは限られた範囲内

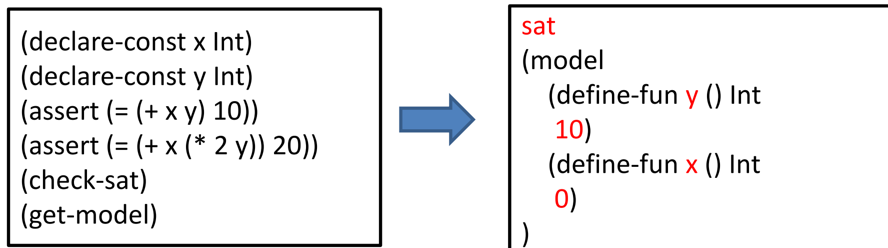

図 2: SMT ソルバによる求解

で網羅的に割り当てられる値の組み合わせを調べるので，検査対象クラスのオブジェクトのフィールドがどのような値であっても満たすべき条件に違反しないかを検査することができる．単純に考えた場合，このような検査を行うにはテストケースを作成して実際にテストを行えばよいが，網羅的にテストケースを作成することや実際にプログラムを実行してテストを行うと多大なコストがかかってしまう．そこで SMT ソルバを利用することでこれらのコストを削減することができる．

2.6 Z3

Z3[19] は Microsoft Research が開発した SMT ソルバの実装である．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．また，SMT-COMP では多くの部門で優秀な成績を収めるなど性能も高い．そのため，多くの研究において Z3 が利用されている．また，Microsoft Research が開発した他のプロダクトの内部エンジンとしても利用されており，表明動的生成系である DySy が内部で使用している Pex も Z3 をエンジンとして利用している．

本研究では SMT ソルバに Z3 を利用する．SMT-LIB 形式で出力することにより，他の SMT ソルバ実装を利用して提案手法を実装することも可能である．

2.7 既存手法

本節では equals メソッドの実装の検査手法として Rupakheti らによって提案されている手法 [9] の概要と課題点について述べる．既存手法では以下の手順で 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．満たすべき規則に違反している場合は，その反例を Alloy Analyzer の視覚的なインスタンス表示機能を用いて出力する．ここで型階層とは，継承関係でつながっている型 (クラスとインタフェース) を階層構造で表したものである．型階層の例を図 3 に示す．Object クラスを除いた継承関係でつながっている型はすべて同じ型階層に含まれるため，この例ではすべてのクラスが 1 つの型階層に含まれている．

1. 型階層に含まれる equals メソッドに対してデータフロー解析を行う

2. equals メソッド中の処理のパターンを解析する

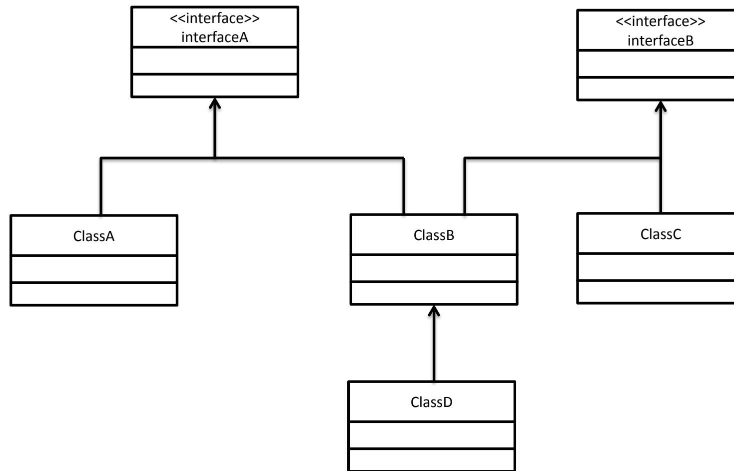

図 3: 型階層の例

3. 型階層と解析した処理のパターンの情報を用いて Java コードを Alloy へ変換する

4. Alloy Analyzer によって検査を行い，反例があればそれを出力する

また，既存手法を実装したツールの概要図を図 4 に示す．このツールは Eclipse プロジェクトを対象としている．まず Java ソースコードの解析ツール Java Development Tools(JDT)[22] を用いて equals メソッドを探す．次に Soot を用いて制御フローグラフを作成し，データフロー解析を行う．次に解析した情報を用いて Alloy のモデルを生成して検査を行う．

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．SMT ソルバの中で Z3 を選択したのはビット演算を扱うことができるからである．hashCode の検査では既存研究の手法をベースとして，新たに解析すべき処理のパターンと SMT ソルバへの入力言語である SMT-LIB への Java コードの変換手法を提案する．

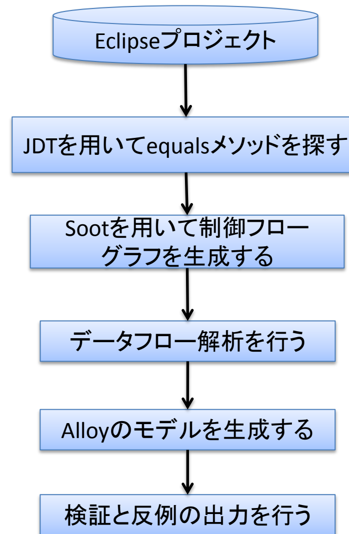

図 4: 既存手法の実装

2.8 関連研究

本節では本研究に関連する既存研究について説明する．

まず，Object クラスのメソッドの設計や実装に関連する既存研究について説明する．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 [23]．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 [24]．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 [25]．このアノテーションによって各クラスの clone メソッドではディープコピーを行うのかシャローコピーを行うのかを示し，統一した実装をサポートすることができる．

次に，モデル検査器や SAT ソルバ，SMT ソルバを用いた検査に関連する既存研究について説明する．Anastasakis らは OCL が付加された UML クラス図から Alloy への変換手法を提案している [26]．この手法を利用することで設計者は Alloy の知識がなくても記述した規則の検査を行うことが可能となる．Liu らはオブジェクト指向言語のコードを SMT ソルバのビットベクトルで表すことに

よってスケーラビリティの高い有界モデル検査を行う手法を提案している [27]．これにより，SAT ソルバを用いた手法よりも高速で検査を行うことが可能となった．Balasubramaniam らはスケーラビリティが高く多くの機能を備えた制約ソルバ MINION を提案している [28]．また，文献 [29] により，解く問題を解析して各問題に最適化された制約ソルバを自動的に生成する手法を提案している．これにより，既存の一体型のソルバよりもメモリ消費量が少なく，実行時間も短いソルバを生成することができた．Burdy らは Java プログラムを静的に検査する手法を提案している [30]．この手法では Java ソースコードを入力として NullPointerException などの例外が発生する可能性がある個所を特定する．また，JML が付加された Java ソースコードの整合性の検査も行うことができる．JML により記述されたメソッドの規則をそれぞれのメソッドが満たしているかを検査することができる．

3　研究動機

本章では、本研究の動機について述べる．

既存研究 [9] では、4 つのオープンソースプロジェクトに対する評価実験により equals メソッドが満たさなければならない規則に違反する実装を発見することができた．これらの規則に違反したクラスの実装は見つけることが難しい欠陥を誘発する原因となる．規則に違反したクラスのオブジェクトをコレクションに格納して使用した場合，正しい振る舞いをせず，データを扱う際にバグが生まれてしまうことがある．例えば，反射性に違反 (a.equals(a) が false を返す) したクラスのオブジェクトをコレクションに格納した場合，contains メソッドでそのオブジェクトがコレクションに含まれているか調べても false が返ってきてしまう．これは List などのコレクションの内部ではオブジェクトの等価性判定に equals メソッドを用いているからである．また，hashCode メソッドについても同様に，equals メソッドで等価と判定されたがハッシュコードの値が異なるオブジェクトを，HashMap や HashSet などのハッシュコードの値をもとにしてデータを格納するコレクションに入れた場合に正しい振る舞いをしなくなってしまう．

図 5 は規則に違反したハッシュコードの実装の具体例で，Apache の PDFBox[31] に対して報告された欠陥である．equals メソッドでは java.util.Arrays の equlas メソッドにより，byte[] 型の配列の要素数と各要素の値が等しいかチェックしているのに対して，hashCode メソッドでは Java.util.Arrays の hashCode メソッドではなく配列 (byte[]) の hashCode メソッドを呼び出してしまっている．配列の hashCode メソッドで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返すため，equals メソッドで等しいと判断されてもインスタンスが異なれば hashCode の値が異なるオブジェクトが存在してしまう．このように異なるインスタンスで，equals メソッドで等価と判定された 2 つのオブジェクトを HashSet に入れようとした場合，ハッシュコードの値が異なるのでどちらも HashSet に格納されてしまう．しかし，この状態では値の重複が許されない Set に等価なオブジェクトが 2 つ入ってしまい正しい振る舞いをしなくなってしまう．このような場合，バグの原因がデータを挿入する処理の部分にあるのか，挿入されるコレクションにあるのか，挿入されるオブジェクト自体にあるのかを調べることはコストがかかってしまう．そこで挿入されるオブジェクトがコレクションに格納されても正しい振る舞いをするか検査することは重要であるといえる．

本研究ではこのような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 メソッドが満たすべき規則に違反しているかどうか検査する手法を提案する．

```
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();
 }
}
```

図 5: hashCode メソッドの規則の違反例

4　提案手法

本章では提案手法について詳細に述べる．

4.1　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．

- 入力: 1 つの型階層
- 出力:規則に違反しているかどうか (違反している場合は反例も出力)

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．

- パスの解析
- equals メソッド，hashCode 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

パスの解析では，検査対象の型階層中のソースコードの制御フローグラフを作成し，データフロー解析を行う．データフロー解析では，ソースコード中のそれぞれの位置で，参照変数がどのクラス

のオブジェクトを指しているのか，どのクラスのメソッドが呼び出されるのかを特定する．これにより特定されたメソッドは，展開が必要な場合に equals メソッドや hashCode メソッド中にインライン展開される．equals メソッド，hashCode メソッド中の処理パターンの解析では，それぞれのメソッド中で行われる処理はほぼパターンが決まっているため，そのパターンのうちのどれに該当するかということを解析する．hashCode 中で行われる処理の中でループによる算術演算とライブラリのメソッド呼び出しは直接的に変換を行うことは難しいのでヒューリスティックスを用いて解析する．SMT-LIB への変換では，処理のパターンの解析で得られた情報をもとにして SMT ソルバへの入力形式である SMT-LIB への変換を行う．この時に検査に必要な制約の追加を行う．本手法では，hashCode メソッドが満たさなければならない 3 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ソースコード解析によって検査を行う．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．また，3 つ目の規則はパフォーマンスをあげるために推奨されている規則であり，必ず満たさなければならないわけではないので本手法では考慮しない．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．SMT ソルバによる検査では，SMT-LIB に変換されたコードを SMT ソルバへ入力して検査を行い，規則に違反しているかどうかが出力される．規則に違反している場合は反例も同時に出力される．

- 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
  (a.equals(b) => a.hashCode()==b.hashCode() (等価規則))

変換例を図 6，7 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，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 7 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．

4.2 パスの解析

パスの解析は基本的に既存手法を利用して行う．パスの解析の手順は以下のとおりである．以降でそれぞれの手順について詳細を述べる．

- 検査対象メソッドの探索
- パス解析

```
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 6: 変換例 (Java)

- パスの刈り込み

4.2.1 検査対象メソッドの探索

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．本手法で検査対象とするメソッドの条件は表 1 の通りである．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親クラスの equals メソッドや hashCode 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．親クラスでもオーバーライドされていない場合はさらに親クラスへと辿って行き，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 を用いて Java バイトコードを解析し，型階層中の各クラスの equals メソッドと hashCode 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．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．

4.2.2 パス解析

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．次にメソッド内のパスそれぞれについてデータフロー解析を行う．データフロー解析では前方解析によって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．次にデー

```
;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) )) ) )
(check-sat)
(get-value (this that other nobj))
```

図 7: 変換例 (SMT-LIB)

表 1: 検査対象のメソッド

メソッド名	引数の数	引数の型	戻り値の型
equals	1	Object	boolean
hashCode	0	-	int

表 2: 展開せずにモデル化を行うメソッド

クラス名	メソッド名
Object	equals()，hashCode()，getClass()
Double	intValue()
Float	floatToIntBits()，intValue()
Arrays	equals()，hashCode()，iterator()，size()
List	equals()，hashCode()，iterator()，size()
Map	equals()，hashCode()，iterator()，size()
Set	equals()，hashCode()，iterator()，size()

タフロー解析の情報を利用してメソッド呼び出しの展開を行う．equals メソッドや hashCode メソッドでメソッド呼び出しが存在する場合は，そのメソッドの振る舞いも考慮しなければ正しくモデル化できないからである．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制限する．本手法で展開するメソッドは検査対象の型階層内に存在するメソッドである．各メソッドは制御フローグラフで表されているため，メソッド呼び出しの部分を展開先のメソッドのパスで置き換える．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．これは前述のように展開を繰り返していくとコストがかかるからである．また hashCode メソッドについて，5 プロジェクト 407 メソッド中の処理を調べたところ，外部メソッドの呼び出しが行われる hashCode メソッドの数は 1 件だけであったため対応しない．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，Collection 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．また，static メソッドは例外的に展開を行う．表 2 に展開せずにモデル化を行うメソッドを示す.

4.2.3　パスの刈り込み

パスの刈り込みでは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．Java では null オブジェクトに対するフィールドの参照やメソッドの呼び出しは null pointer exception を発生させてしまうため，その可能性があるパスは刈り込みを行う．このパスが存在するメソッドにはエラーを出力する．また，例外を投げる throw が存在するパスも刈り込みを行う．これは例外が起こるパスのモデル可が難しいからである．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．

4.3　equals メソッドと hashCode メソッド中の処理のパターンの解析

処理のパターンの解析では equals メソッドと hashCode メソッド中で行われている処理がどのパターンに該当するかを解析する．それぞれのパターンに対してどのようにモデル化するかが決まっており，SMT-LIB への変換の際にこの情報を利用する．また，メソッド中で行われている処理のパターンの解析に加えて，このステップではサブセット規則に違反しているかどうかの判定も行う．以降でそれぞれの手順について詳細に述べる．

4.3.1　フィールドのサブセット判定

フィールドのサブセット判定では hashCode メソッドで使用されているフィールドの集合が equals メソッドで使用されているフィールドの集合のサブセットであるかの判定を行う．equals メソッドと hashCode メソッドをそれぞれ解析して，使用されているフィールドの集合を求め，サブセットになっているかを調べる．親クラスのメソッドや同じクラスの他のメソッドを呼び出している場合，パス解析の時点でそれらのメソッドは検査対象の equals メソッドや hashCode メソッド内に展開されているため，親クラスや他のメソッド中で使用されているフィールドも集合の中に含まれる．親クラスや他のメソッドで使用されているフィールドの値が変化した場合，equals メソッドや hashCode メソッドの戻り値に影響を及ぼすのでこれらのフィールドも考慮すべきであると考えられる．本手法では hashCode メソッドが満たさなければならない規則の 1 つをサブセット規則で代用した．これは，一般的に equals メソッドで使用される情報が変更されなければハッシュコードは同じ値を返すという規則を 2 つのケースに分けた場合のうちの 1 つである．1 つは本手法で検査している hashCode メソッドでは equals メソッドで使用しているフィールドのみを使用しているというケースである．このケースでは，hashCode メソッドで使用しているフィールドは equals メソッドで使用されているものだけであるので，equals メソッドで使用されている情報が変更されなければハッシュコードも変化しない．2 つ目は hashCode メソッドにおいて，equals メソッドで使用されているフィールドとは異なるフィールドを使用しているケースである．このケースでは，equlas メソッドで使用されている

情報が変更されなくても，hashCode メソッドのみで使用しているフィールドなどの値が変化した場合，ハッシュコードが変化してしまう可能性が高い．これを検査するためには equals メソッドで使用されているフィールドと hashCode メソッドで使用しているフィールドの値の関係を調べなければならない．この関係を調べるためにはフィールドの値を変化させる全てのメソッドを解析する必要があり，コストがかかりすぎるため検査することは難しい．

4.3.2　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メソッド呼び出し，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．

4.3.3　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，ループを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．ハッシュコードメソッドで行われるループ演算はパターンが決まっており，本手法ではそのパターンに該当するもののみの変換を行う．詳細については 4.4 節で述べる．

4.4　SMT-LIB への変換

SMT-LIB への変換は 2 つのステップから成り立っている．基本的な構造の変換では，クラスやフィールド，継承関係，メソッドなどの Java 言語を構成する基本的な構造をどのようにして SMT-LIB で表すかについて述べる．メソッド中で行われている処理の変換では equals メソッドと hashCode メソッド中で行われている処理をどのように SMT-LIB で表すかについて述べる．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用いて変換を行う．

4.4.1　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．SMT-LIB のレコード定義では複数のデータ型の値を 1 つにまとめた新たなデータ型を定義する．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．クラスのフィールドは equals メソッドか hashCode メソッドで使用されている変数のみが定義される．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．列挙型のフィールドは SMT-LIB の列挙型の定義を用いて表現する．列挙型の参照変数は null を指している場合もあるため，列挙型で定義されている識別子に NULL を追加してモデル化を行う．また，列挙型の hashCode メソッドでは Object クラスの hashCode メソッドが呼び出されるため識別子ごとに異なる値を返すようにモデル化を行う．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を満たすように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．Ref 型オブジェクトを使用する場合は Ref 型オブジェクトの equals メソッドも同時に定義する．Ref 型オブジェクトの hashCode メソッドは定義せず，フィールドのハッシュコードの値を参照するようにモデル化を行う．また，以下のような Java のオブジェクトに成り立つ性質と，equals メソッドと hashCode メソッドの満たすべき規則を満たすように実装されているオブジェクトに成り立つ制約も記述する．

- Ref 型のオブジェクトはポインターの値が等しければ全てのフィールドの値が等しい
- 抽象クラスとインタフェースはオブジェクト生成できない (実行時の型は抽象クラスとインタフェース以外の型でクラスである)
- equals メソッドで等価と判定された Ref 型のオブジェクトのハッシュコードは等しい

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．

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．これにより親クラスのフィールド参照をしているメソッドをモデル化することができる．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．図 8 に型の列挙と instanceof 演算のモデル化例を示す．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．また，Ref 型と同様に Java のオブジェクトに成り立つ性質である以下の 2 つの制約を記述する．記述例を図 9 に示す．

- Object 型のオブジェクトはポインターの値が等しければ全てのフィールドの値が等しい
- ポインターの値が null(0) ならば実行時の型を表すフィールドの値は NULL である

本手法で検査する規則は，最大で 3 つのオブジェクト間に成り立つ関係を表しているため，値の割り当てを求める変数として Object 型の変数を this，that，other という名前で宣言する．これらの変数は equals メソッドの反射性，対称性，推移性と hashCode メソッドの等価規則を検査する対象のオブジェクトであるため，ポインターの値は null(0) でないという制約を記述する．また”null でない任意の参照値 x について x.equals(nul) は false を返す”という規則の検査のためにポインターの値が null である Object 型の変数 nobj の宣言も行う．記述例を図 9 に示す．

4.4.2 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．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．この情報を用いて各文を 2 分木によって表したものが式木である．変換は各文に対して作成された式木の情報を用いて行う．変換の方針としては return 文で返している式からソースコードを逆順にさかのぼっていく．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を表 3 に示す．Java コードから SMT-LIB の変換関数を μ とし，以下では型 boolean，数値型を持つ部分式をそれぞれ b_m，n_m で表す．また，任意の型を持つ部分式を a_m，任意の型を T_m で表す．Java では基本的に中置記法によって式を記述するが，SMT-LIB では前置記法によって式を記述する．また，instanceof

```
(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)) )))
```

図 8: incetaneof 演算のモデル化

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．

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である instanceof，getclass()，.class に関しては表 3 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 3 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う．ここでは配列を例として具体的な説明を行う．まず配列が解析対象クラスのフィールドであり，equals メソッドを実行しているオブジェクトの配列と，引数で渡されたオブジェクトの配列に対して同じインデックスを用いて比較を行っているかを調べる．次にループヘッダで使用されている変数が配列のインデックスとして使用されているかを調べる．これらの条件を満たしている場合は配列の比較を行っているとみなし，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．

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．

```
(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)
(assert (iff (= (pointer this) (pointer that)) (= this that)))
(assert (iff (= (pointer this) (pointer other)) (= this other)))
(assert (iff (= (pointer that) (pointer other)) (= that other)))
(assert (> (pointer this) 0))
(assert (> (pointer that) 0))
(assert (> (pointer other) 0))
(assert (= (pointer nobj) 0))
(assert (iff (= (pointer this) 0) (= (class this) Null)))
(assert (iff (= (pointer that) 0) (= (class that) Null)))
(assert (iff (= (pointer other) 0) (= (class other) Null)))
(assert (iff (= (pointer nobj) 0) (= (class nobj) Null)))
(assert (not(= (class this) ArConstants)) )
(assert (not(= (class that) ArConstants)) )
(assert (not(= (class other) ArConstants)) )
(assert (=> (= (pointer(filename (ar this))) (pointer(filename (ar that))))
(= (filename(ar this)) (filename(ar that)) )  ))
(assert (=> (= (pointer(filename (ar other))) (pointer(filename (ar that))))
(= (filename(ar other)) (filename(ar that)) )  ))
(assert (=> (= (pointer(filename (ar this))) (pointer(filename (ar other))))
(= (filename(ar this)) (filename(ar other)) )  ))
(assert (=> (= (eqnum(filename (ar this))) (eqnum(filename (ar that)))) (=
(hsnum(filename(ar this))) (hsnum(filename(ar that))) )  ))
(assert (=> (= (eqnum(filename (ar this))) (eqnum(filename (ar other)))) (=
(hsnum(filename(ar this))) (hsnum(filename(ar other))) )  ))
(assert (=> (= (eqnum(filename (ar other))) (eqnum(filename (ar that)))) (=
(hsnum(filename(ar other))) (hsnum(filename(ar that))) )  ))
```

図 9: オブジェクトに成り立つ制約の記述例

表 3: μ 変換：単純な変換が可能な演算

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

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 つの変数の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，8 ビットのモデル化でも範囲は制限されてしまうが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．そのような動的に式を評価する仕組みは SMT ソルバには存在しない．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．しかし，違反してない場合にそのコードが確実に正しいということはできない．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．

4.4.3　検査条件の挿入

本手法では equals メソッドが満たさなければならない 4 つの規則と，hashCode メソッドが満たさなければならない 2 つの規則のうちの等価規則に違反しているかどうかを SMT ソルバにより検査する．SMT ソルバは SMT-LIB で記述した制約を充足する解を求めるため，満たさなければならない規則の否定を満たす解が存在するかどうか，つまり反例が存在するかどうかを求める．

4.5　SMT ソルバによる検査

SMT ソルバによる検査では SMT ソルバ Z3 を用いて SMT-LIB に変換された Java コードの検査を行う．検査項目は equals メソッドが満たさなければならない 4 つの規則と，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 つである．サブセット規則の検査はソースコード解析の時点で行うので SMT ソルバによる検査は行わない．SMT-LIB に変換されたコードを Z3 に与えると，それぞれの規則について規則に違反しているかどうかと反例がある場合はその反例を出力する．Z3 では反例がない場合は unsat と出力され，これは SMT-LIB で記述された制約を満たす解がないことを示す．本手法ではそれぞれのメソッドが満たさなければならない規則の否定を制約として与えているので unsat が出力された場合は満たすべき規則に違反していないことになる．sat が出力された場合は満たすべき規則に違反していることになり，反例として違反している場合の変数の値が出力される．

Z3 による検査結果の例を図 10 に示す．これは図 7 のコードを Z3 によって検査した結果である．上から 4 つが equals メソッドの満たさなければならない規則の検査結果であり，5 つ目が hashCode のメソッドの等価規則の検査結果である．この例では equals メソッドが満たさなければならない規則の違反は検出されず，hashCode メソッドが満たさなければならない規則の違反が検出されている．反例では 2 つの ArEntry オブジェクトが存在し，フィールドで保持している 2 つのオブジェクトは等価だが参照は等しくないという状態が示されている．ArEntry の hashCode メソッドでは親クラスの hashCode メソッドを呼び出しているが，親クラスでは hashCode メソッドをオーバーライドしていないため，Object クラスの hashCode メソッドを呼び出すことになる．結果として，研究動機の例と同様の理由で equals メソッドで等価と判定されてもハッシュコードが異なってしまう．

SMT ソルバは基本的に，記述された制約を充足する解が存在するかどうかを調べることを目的としており，全ての充足する解を求めることを目的としていない．よって，充足する解が存在する場合，そのうちの 1 つしか求められない．本手法では，規則に違反しているかどうかを検査することを目的としているため，出力される反例は 1 つで十分であるが他の反例が存在するかどうかも調べる

```
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)))
```

図 10: Z3 による検査結果

ことは可能である．反例として出力された変数のうちランダムに 1 つの変数の値を否定するような制約を追加し，もう一度 SMT ソルバに求解させれば最初に得られた反例とは違う反例が得られる．最初に得られた反例の他に反例が存在しない場合は unsat が出力される．これを繰り返すことによって全ての反例を求めることが可能である．

表 4: ツールの適用結果

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
lucene4.6.0	110	106	4

5 評価と考察

5.1 評価の概要

本章では本手法の有用性の評価のために行ったケーススタディについて述べる．本研究ではサブセット規則の検査を行うツールを実装した．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サブセット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
- 規則に違反していない実装を間違って違反だと検出しないか

5.2 評価 1:サブセット規則について

5.2.1 結果

ツールを lucene4.6.0[32] に対して適用した結果を表 5.2.1 に示す．対象クラス数は検査対象のクラス数を表しており，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を対象としている．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を表している．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．

5.2.2 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 System.identityHashCode() を使用してオブジェク

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している．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，親クラスでもオーバライドされていないため，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．

また，検査結果を手動で確認し，違反しているクラスの検出が正しく行えていることを確認できた．

5.3 評価2:等価規則について

実験2ではApache Hive[33]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．修正前のコードが規則違反と検出され，修正後のコードは規則違反ではないと検出されれば，本手法により実プロジェクトの中で規則に違反している実装を検出することが可能であることを確認できる．

修正前のコードを図11に示す．また，修正により追加されたhashCodeメソッドを図12に示す．getTypeStringメソッドはゲッターである．このクラスには親クラスが存在せず，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．

図13に修正前のコードをSMT-LIBに変換したモデルの一部を示す．また，図14に修正後のコードをSMT-LIBに変換したモデルの一部を示す．2つのモデルのequalsMain関数はequalsメソッドをモデル化したものである．この関数は，元のJavaコードに従って対象のオブジェクトやフィールドのnullチェック，フィールドの値の比較を行うようにモデル化されている．修正前のコードと修正後のコードでequalsメソッドは変化していないのため，両方のモデルに同じequalsMain関数が出力されている．修正前のコードのhashCodeメソッドはObjectクラスから継承したものであるため，修正前のコードを変換したモデル中のhashCode関数は，ObjectクラスのhashCodeメソッドの振る舞いをモデル化したものになっている．よって，2つのオブジェクトが等価と判定されても，オブジェクトが異なればハッシュコードが異なり，等価規則に違反してしまう．修正後のコードを変換したモデル中のhashCode関数は，オーバーライドされたhashCodeメソッドの振る舞いをモデル化したものになっている．オーバーライドされたhashCodeメソッドではフィールドの値を使用してハッシュコードを計算しているためequalsメソッドで等価と判定されたオブジェクトのハッシュコードは等しくなる．

これらのモデルに対して 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．

```
public class HCatFieldSchema implements Serializable {

  public enum Category {
    PRIMITIVE,ARRAY,MAP,STRUCT
  };

  String fieldName;
  Category category ;
  String typeString;

  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;
  }

  //not override hashCode()
}
```

図 11: 修正前のコード

```
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;
}
```

図 12: 修正により追加された hashCode メソッド

```
;CHA information
(declare-datatypes () ((Type HCatFieldSchema Serializable UnderSeri Object
Null)) )
...
(declare-datatypes () ((Category PRIMITIVE ARRAY MAP STRUCT NULL)) )
(declare-datatypes () (( Ref(Rfield (eqnum Int) (hsnum Int) (pointer Int)) )) )
(declare-datatypes () ((HCatFieldSchema(Hfield (category Category)(fieldname
Ref)(typeString Ref)) )) )
(declare-datatypes ()  (( Object(Ofield (hcat HCatFieldSchema)(pointer
Int)(class Type)))))
...
(define-fun equalsMain ((o1 Object)(o2 Object)) Bool
 (and (=> (or (or (= (class o1) Serializable )(= (class o1) UnderSeri)) (=
(class o1) Object)) (= (pointer o1)(pointer o2)))
  (=> (= (class o1) HCatFieldSchema)
   (ite (= (pointer o1) (pointer o2)) true
    (ite (= (pointer o2) 0) false
     (ite (not(instanceof (class o2) HCatFieldSchema)) false
      (ite (not(= (category(hcat o1)) (category(hcat o2)))) false
       (ite (and (= (pointer(fieldname(hcat o1))) 0) (not(=
                        (pointer(fieldname(hcat o2))) 0))) false
        (ite (not(equalsRef (fieldname(hcat o1)) (fieldname(hcat o2)))) false
         (ite (and (= (pointer(typeString(hcat o1))) 0) (not(=
                      (pointer(typeString(hcat o2))) 0))) false
          (ite (not(equalsRef (typeString(hcat o1)) (typeString(hcat o2))))
                        false true)
         )
        )
       )
      )
     )
    )
   )
  )
 )
)

(define-fun hashCode ((o1 Object)) Int
  (pointer o1)
)
...
```

図 13: 変換された修正前のコード

```
;CHA information
(declare-datatypes () ((Type HCatFieldSchema Serializable UnderSeri Object
Null)) )
...
(declare-datatypes () ((Category PRIMITIVE ARRAY MAP STRUCT NULL)) )
(declare-datatypes () (( Ref(Rfield (eqnum Int) (hsnum Int) (pointer Int)) )) )
(declare-datatypes () ((HCatFieldSchema(Hfield (category Category)(fieldname
Ref)(typeString Ref)) )) )
(declare-datatypes ()  (( Object(Ofield (hcat HCatFieldSchema)(pointer
Int)(class Type)))))
...
(define-fun equalsMain ((o1 Object)(o2 Object)) Bool
 (and (=> (or (or (= (class o1) Serializable )(= (class o1) UnderSeri)) (=
(class o1) Object)) (= (pointer o1)(pointer o2)))
  (=> (= (class o1) HCatFieldSchema)
   (ite (= (pointer o1) (pointer o2)) true
    (ite (= (pointer o2) 0) false
     (ite (not(instanceof (class o2) HCatFieldSchema)) false
      (ite (not(= (category(hcat o1)) (category(hcat o2)))) false
       (ite (and (= (pointer(fieldname(hcat o1))) 0) (not(=
                          (pointer(fieldname(hcat o2))) 0))) false
        (ite (not(equalsRef (fieldname(hcat o1)) (fieldname(hcat o2)))) false
         (ite (and (= (pointer(typeString(hcat o1))) 0) (not(=
                            (pointer(typeString(hcat o2))) 0))) false
          (ite (not(equalsRef (typeString(hcat o1)) (typeString(hcat o2))))
                              false true)
         )
        )
       )
      )
     )
    )
   )
  )
 )
)

(define-fun hashCode ((o1 Object)) Int
 (ite (= (class o1) HCatFieldSchema)
  (+ (ite (= (pointer(typeString(hcat o1))) 0) 0 (hsnum (typeString(hcat
o1))) )(* 31
   (+ (ite (= (pointer(fieldname(hcat o1))) 0) 0 (hsnum (fieldname(hcat
o1))) )(* 31
    (+ (ite (= (category(hcat o1)) NULL) 0 (hashCodeEnum (category(hcat
o1))) )(* 31 17 ))
   ))
  ))
  (pointer o1)
 )
)
...
```

図 14: 変換された修正後のコード

6　あとがき

本研究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，提案手法の全自動化ツールの作成や，様々なプロジェクトに対する評価実験があげられる．本研究ではケーススタディによ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．また，Z3 ではビットベクトルから Int 型への変換に時間がかかるため，Java の int 型の長さである 32 ビットのビットベクトルを用いたモデル化による検査を実時間で行うことができない．hashCode メソッドではビット演算が使用されるため，この問題にも対応する必要がある．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究の全過程を通して， 熱心かつ丁寧なご指導を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究に関して，厳しくも的確なご助言を頂きました 井垣 宏 特任准教授 に深く感謝申し上げます.

本研究を行うにあたり，日常の議論の中でご助言を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

その他の楠本研究室の皆様のご助言，ご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等で多くの知識や示唆を頂きました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[1] Joshua 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] David Hovemeyer and William Pugh. "Finding bugs is easy". In *ACM SIGPLAN Notices Homepage archive*, pp. 92–106, 2004.

[4] Julian Dolby, Mandana Vaziri, and Frank Tip. "Finding bugs efficiently with a SAT solver". In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 195–204, 2007.

[5] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. "Declarative Object Identity Using Relation Types". In *Proceedings of the 21st European Conference on Object-Oriented Programming*, pp. 54–78, 2007.

[6] Chandan R. Rupakheti and Daqing Hou. "An Empirical Study of the Design and Implementation of Object Equality in Java". In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, pp. 111–125, 2008.

[7] Chandan R. Rupakheti and Daqing Hou. "An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java". In *Proceedings of the 17th Working Conference on Reverse Engineering*, pp. 205–214, 2010.

[8] Chandan R. Rupakheti and Daqing Hou. "EQ: Checking the Implementation of Equality in Java". In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pp. 590–593, 2011.

[9] Chandan R. Rupakheti and Daqing Hou. "Finding Errors from Reverse-Engineered Equality Models using a Constraint Solver". In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pp. 77–86, 2012.

[10] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. "The Soot framework for Java program analysis: a retrospective". In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop*, 2011.

[11] Daniel Jackson. "Alloy: a lightweight object modelling notation". *ACM Transactions on Software Engineering and Methodology*, Vol. 11, No. 2, pp. 256–290, 2002.

[12] Daniel Jackson. *"Software abstractions: logic, language, and analysis"*. The MIT Press, 2011.

[13] Daniel Jackson. " Alloy: a language and tool for relational models", 2013. http://alloy.mit.edu.

[14] 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.

[15] Bruno Dutertre and Leonardo de Moura. "A Fast Linear-Arithmetic Solver for DPLL(T)". In *Proceedings of the 18th international conference on Computer Aided Verification*, pp. 81–94, 2006.

[16] Clark Barrett and Cesare Tinelli. "CVC3". In *Proceedings of the 19th International Conference on Computer Aided Verification*, pp. 298–302, 2007.

[17] Alberto Griggio. "A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic". *JSAT*, Vol. 8, pp. 1–27, 2012.

[18] Alessandro Cimatti, Alberto Griggio, Bastiaan J. Schaafsma, and Roberto Sebastiani. " The MathSAT5 SMT Solver". In *Proceedings of the 19th international conference on Tools and algorithms for the construction and analysis of systems*, pp. 93–107, 2013.

[19] Leonardo de Moura and Nikolaj Bjorner. "Z3: An Efficient SMT Solver". In *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems*, pp. 337–340, 2008.

[20] D. Cok, A. Stump, and M. Deters. "SMT-COMP2012", 2012. http://smtcomp.sourceforge.net/2012/.

[21] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard Version 2.0", 2012. http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf.

[22] the Eclipse Foundation. " Eclipse Java development tools (JDT)", 2013. http://www.eclipse.org/jdt/.

[23] Derek Rayside, Zev Benjamin, Rishabh Singh, Joseph P. Near, Aleksandar Milicevic, and Daniel Jackson. "Equality and Hashing for (almost) Free: Generating Implementations from Abstraction Functions". In *Proceedings of the 31st International Conference on Software Engineering,*, pp. 342–352, 2009.

[24] Neville Grech, Julian Rathke, and Bernd Fischer. "JEqualityGen: Generating Equality and Hashing Methods". In *Proceedings of the ninth international conference on Generative programming and component engineering*, pp. 177–186, 2010.

[25] Thomas Jensen, Florent Kirchner, David Pichardie, and Inria Rennes Bretagne Atlantique. "Secure the clones: Static enforcement of policies for secure object copying". Technical report, 2010.

[26] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. "UML2Alloy: A Challenging Model Transformation". In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, pp. 436–450, 2007.

[27] Tianhai Liu, Michael Nagel, and Mana Taghdiri. "Bounded Program Verification using an SMT Solver: A Case Study". In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pp. 101–110, 2012.

[28] Ian P. Gent, Chris Jefferson, and Ian Miguel. "Minion: A Fast, Scalable, Constraint Solver". In *Proceedings of the 17th European Conference on Artificial Intelligence*, pp. 98–102, 2006.

[29] Dharini Balasubramaniam, Christopher Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. "An Automated Approach to Generating Efficient Constraint Solvers". In *Proceedings of the 2012 International Conference on Software Engineering*, pp. 661–671, 2012.

[30] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. "An overview of JML tools and applications". *International Journal on Software Tools for Technology Transfer*, pp. 212–232, 2005.

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

[32] Apache. "Apache Lucene - A Lucene Core", 2012. http://lucene.apache.org/core/.

[33] Apache. "Apache Hive TM", 2014. http://hive.apache.org/.