特別研究報告

題目

Java における equals メソッドと hashCode メソッドの検査ツールの実装と評価

指導教員
楠本 真二 教授

報告者
尾ノ上 博樹

平成 26 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 25 年度 特別研究報告

Java における equals メソッドと hashCode メソッドの検査ツールの実装と評価

尾ノ上 博樹

内容梗概

Java において equals メソッドと hashCode メソッドは Object クラスで定義されており，コレクションフレームワークに格納されるようなデータ構造を表すクラスにおいて重要である．原則として，equals メソッドをオーバーライドする場合は hashCode メソッドもオーバーライドしなければならない．これらのメソッドには満たすべき規則が存在し，その規則を満たさない実装を行った場合，本来それらのメソッドに期待されている振舞いが保証されなくなる．さらには，想定外の振舞いによって発見の難しい欠陥を誘発する可能性がある．

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を満たしているかを軽量形式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 を用いてモデル化し Alloy Analyzer によって検査を行っている．しかし，一般に equals メソッドがオーバーライドされている場合は hashCode メソッドも同時にオーバーライドされているため，両方のメソッドを検査すべきである．

著者の属する研究グループでは equals メソッドだけでなく hashCode メソッドも対象とした検査手法が提案されている．この手法では hashCode メソッドを検査するために Alloy ではなく SMT-LIB を用いてモデル化を行い，SMT ソルバの 1 つである Z3 によって検査を行う．

本研究ではこの手法の一部を Eclipse プラグインとして実装し，いくつかのオープンソースプロジェクトに対して適用し評価を行った．結果として，equals メソッドや hashCode メソッドの規則を満たさない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

図 目 次

表 目 次

1　はじめに

Java において equals メソッドと hashCode メソッドは Object クラスで定義されており [1]，コレクションフレームワークに格納されるようなオブジェクト同士の等価判定を行うデータ構造を表すクラスにおいて重要である．原則として，equals メソッドをオーバーライドする場合は hashCode メソッドもオーバーライドしなければならない．これらのメソッドには Oracle の API 規則によって満たすべき規則が定められており [2]，その規則を満たさない実装を行った場合，本来それらのメソッドに期待されている振舞いが保証されなくなる．さらには，想定外の振舞いによって発見の難しい欠陥を誘発する可能性がある [1][3][4][5]．

equals メソッドのみを対象として満たすべき規則を満たしているかを検査する手法として，Rupakheti らの検査手法が知られている [6][7][8][9]．この手法の処理の流れは，まず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 Analyazer 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ないか検査を行う．しかし，一般に equals メソッドがオーバーライドされている場合は hashCode メソッドも同時にオーバーライドされているため，両方のメソッドを検査すべきである．また，Alloy にはビット演算が存在しないためビット演算が存在するメソッドの検査も正しく行われていない．

著者の属する研究グループでは equals メソッドだけでなく hashCode メソッドも対象とした検査手法が提案されている [10]．この手法では hashCode メソッドを検査するために Alloy ではなく SMT-LIB によるモデル化を行い，SMT ソルバの 1 つである Z3 によって検査を行う．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には違いがあるため，変換する hashCode メソッドの代表的な処理のパターンと Java コードの SMT-LIB への変換方法が新たに提案されている．

そこで，本研究ではこの手法の一部を Eclipse プラグイン形式のツールとして実装した．さらに，いくつかのオープンソースプロジェクトに対してツールを適用し評価を行った．その結果，equals メソッドや hashCode メソッドの満たすべき規則に違反した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，著者の属する研究グループで提案されている手法について説明する．4 章では手法をツールとして実装した詳細について述べ，5 章ではツールを用いて行った実験について述べる．最後に，6 章で本研究のまとめを述べる．

2　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 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[11] 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

```
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

- 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[12] 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 [13] 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[14] 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 [15] 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[15] 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．

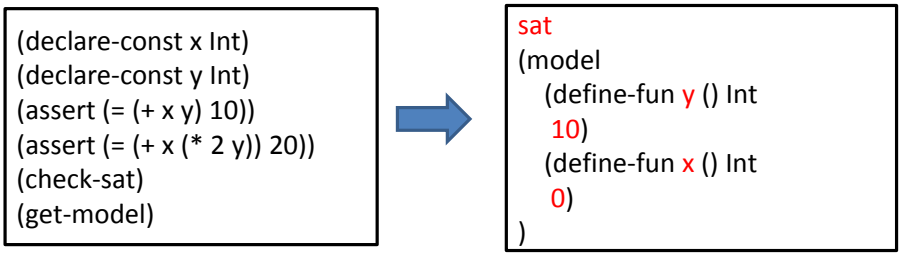

図 2: SMT ソルバによる求解

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 [16][17][18][19][20]，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 [21] 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[22] 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[20] を利用する．SMT ソルバは限られた範囲内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[20] 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
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)[23] 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

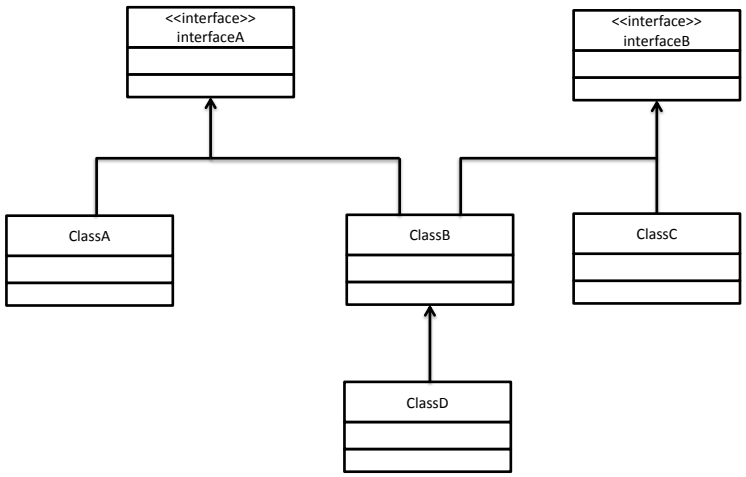

図 3: 型階層の例

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．

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 [24]．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 [25]．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 [26]．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

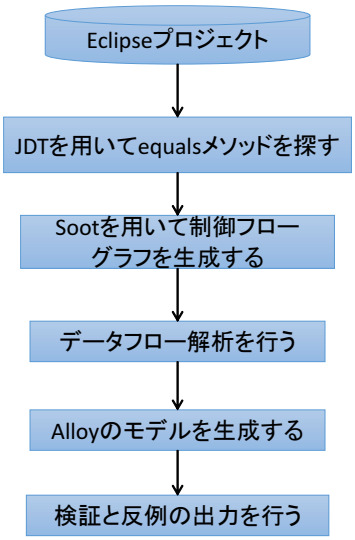

図 4: 既存手法の実装

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 [27]．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 [28]．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 [29]．また，文献 [30] 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 [31]．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．

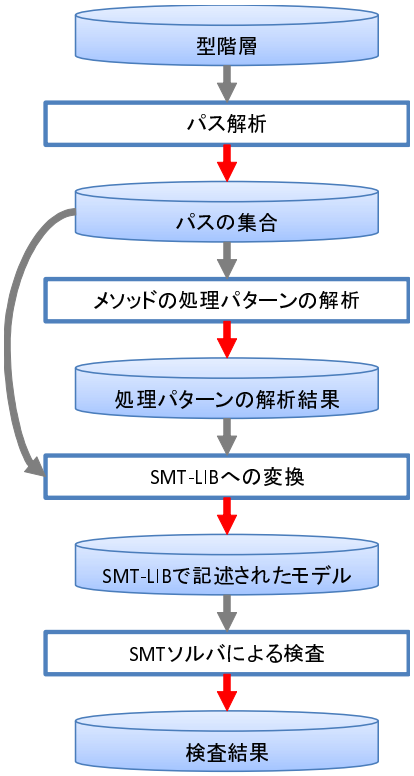

図 5: 提案手法の処理の流れ

3 提案手法

本章では，著者の研究グループで提案されている手法について説明する．

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 メソッドおよび hashCode メソッドの振舞いを SMT-LIB を用いてモデル化し，SMT-LIB で記述されたモデルを SMT ソルバの 1 つである Z3 により検査する．

提案手法の処理の流れを図 5 に示す．円盤状のノードは資源を，四角のノードは処理を表している．赤い矢印は出力の流れを，黒い矢印は入力の流れを表している．提案手法は 1 つの型階層を入力とし equals メソッドおよび hashCode メソッドがそれぞれの規則を満たしているかどうかを出力する．

図 5 に示す通り，提案手法は大きく 4 つのステップから成り立つ．パス解析では対象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．メソッドの処理パターンの解析では equals メソッドや hashCode メソッド中で行われる処理において，事前に定めておいた複数のパターンとのマッチングを行う．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．SMTソルバによる検査ではSMT-LIBで記述されたモデルを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を出力する．規則に違反している場合は反例も同時に出力する．

提案手法ではequalsメソッドの満たすべき規則については変更なく扱うが，hashCodeメソッドの満たすべき規則については一部変更して扱う．2.1.2節で述べた通り，hashCodeメソッドの満たすべき規則は以下の通りである．

- 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．ただし，この整数は同じアプリケーションの実行ごとに同じである必要はない．

- 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．

- equals(java.lang.Object)メソッドで2つのオブジェクトが等価でないとされた場合は，これらのオブジェクトに対してhashCodeメソッドを呼び出したときに，結果が異なる整数値にならなくてもかまわない．しかし，等しくないオブジェクトについては異なる整数値が生成されるようにすれば，ハッシュテーブルのパフォーマンスを上げることができる．

3つ目の規則は，推奨されている規則であり必ずしも満たしている必要はないので提案手法では検査しない．2つ目の規則は変更なくこのまま検査する．また，以降でこの規則を等価規則と呼ぶ．

1つ目の規則は時間の概念が含まれておりSMT-LIBでモデル化することが困難なため，提案手法では以下のサブセット規則を用いて検査を行う．

- hashCodeメソッドで参照されているフィールドの集合は，equalsメソッドで参照されているフィールドのサブセットである．

次節以降で各ステップの詳細について述べる．

3.2 パス解析

パス解析では型階層を入力としメソッドの実行経路であるパスの集合を出力する．このパスは，equalsメソッドおよびhashCodeメソッドそれぞれについて，1つも存在しないこともあれば複数存在することもある．

手順は検査対象メソッドの探索，メソッド内パス解析とメソッド間パス解析，パスの刈り込みの 3 つのステップからなる．次小節以降でそれぞれのステップについて述べる．

3.2.1 検査対象メソッドの探索

検査対象メソッドの探索では型階層を入力とし制御フローグラフを出力する．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．これは，表 1 のようなメソッド名，引数の数およびその型，戻り値の型を持つメソッドを探すことで行う．equals メソッドをオーバーライドしていないクラスは，equals メソッドをオーバーライドしている直近の祖先クラスの equals メソッドをそのクラスの equals メソッドとする（すなわち equals メソッドの継承）．型階層中にオーバーライドしている祖先クラスが存在しなければ，Object クラスの equals メソッドを継承する．hashCode メソッドをオーバーライドしていないクラスに関しても同様に扱う．次に，Soot を用いて Java バイトコードを解析し，equals メソッドおよび hashCode メソッドの制御フローグラフを得る．この制御フローグラフは Jimple 形式の中間コードで表現されており，以降では Jimple コードを対象として解析を行う．

3.2.2 メソッド内パス解析とメソッド間パス解析

メソッド内パス解析とメソッド間パス解析では，制御フローグラフを入力としパスを出力する．

まず，メソッド内パス解析を行い制御フローグラフからメソッド内パスを得る．メソッド内パスとは，メソッド呼出しの展開を行っていないあるメソッドの中のみで閉じたパスのことである．

次に，メソッド間パス解析を行いメソッド内パスからパスを得る．メソッド間パス解析ではメソッド内パスにおいてメソッド呼出しを行っているノードを，呼び出し先のメソッドの制御フローグラフで置換する（メソッド呼出しの展開）．しかし，時空間的なコストを削減するため以下に挙げるメソッド呼出しは展開しない．

ゲッターメソッド this.getState() == that.getState() のような比較の中で使われているゲッターメソッドは展開せずに，SMT-LIB でモデル化する際に，(= (state this)(state that)) のようにゲッターメソッドの返り値として得られる変数の比較へと変換する．

表 1: メソッドの検出条件

対象メソッド	メソッド名	引数の数	引数の型	返り値の型
equals メソッド	equals	1	java.lang.Object	boolean
hashCode メソッド	hashCode	0	-	int

コレクションフレームワークのメソッド size メソッドなど振舞いが既知であるメソッドをあらかじめ定義された関数としてモデル化する．

ラッパークラスのメソッド intValue メソッドなど振舞いが既知であるメソッドをあらかじめ定義された関数としてモデル化する．

Object クラスのメソッド getClass メソッドなど振舞いが既知であるメソッドをあらかじめ定義された関数としてモデル化する．

上記以外の型階層外のメソッド 型階層外のクラスのメソッドは展開せずに，非決定性関数としてモデル化する．

フィールドに対するメソッド呼出し 既存研究において実装を単純化するために展開されていないため，提案手法でも展開しない．

3.2.3 パスの刈り込み

パスの刈り込みでは，パス解析で得られたパスの集合に対し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．また，例外を投げる throw 文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すべきパスを取り除くことでパフォーマンスを向上させることができる．

3.3 メソッドの処理パターンの解析

メソッドの処理パターンの解析では，パスの集合中のパスを入力とし 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解析する．また，フィールドのサブセットの判定も行う．次小節以降でフィールドのサブセット判定とそれぞれのメソッドの処理パターンの解析の詳細について述べる．

3.3.1 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 メソッドで参照されているフィールドの集合が equals メソッドで参照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．

それぞれのメソッド中で，同じクラス内もしくは祖先クラスのメソッドを呼び出しており，かつそのメソッド呼出しが展開されている場合は，そのメソッド中で参照されているフィールドも集合に含める．この判定は，3.1 節で述べた hashCode メソッドのサブセット規則を検査していることになる．

3.3.2 equals メソッドの処理パターンの解析

equals メソッドの処理パターンの解析では，特定の処理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 つである．型のチェックは，if 文の条件式中での instansof 演算子や getClass メソッドを用いた判定，型のキャストを行っている処理パターンのことである．状態のチェックは，フィールドが null でないか判定を行っている処理パターンを指す．配列，List，Set，Map の等価判定は，ループを用いてそれぞれの持つ要素の比較を行っている処理パターンである．

3.3.3 hashCode メソッドの処理パターンの解析

hashCode メソッドの処理パターンの解析では，equals メソッドの処理パターンの解析と同様に特定の処理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，int 型への変換，ビット演算，ループによる算術演算の 3 つである．int 型への変換は，キャストやメソッドを用いた int 型への変換を行っている処理パターンのことである．ビット演算は，プリミティブ型に対して XOR などのビット演算を行っている処理パターンを指す．ループによる算術演算では，ループを用いてデータ構造の要素同士の算術演算を行っているかの解析を行う．

3.4 SMT-LIB への変換

SMT-LIB への変換では，処理パターン解析の結果およびパスを入力とし SMT-LIB で記述されたモデルを出力する．Java コードを基本的な構造部分とメソッド部分に二分し，それぞれについて SMT-LIB への変換を行う．基本的な構造とはクラスやフィールド，継承関係などである．Java コードを SMT-LIB でモデル化した後は，満たさなければならない規則を SMT-LIB の assert 制約式を用いて表したものを追加する．次小節以降で，基本的な構造の変換とメソッドの変換について詳しく述べる．

3.4.1 基本的な構造の変換

Java における基本的な構造を SMT-LIB でどのように表すかについて述べる．クラスは SMT-LIB のレコード定義を用いてどのようなフィールドを持つかを表現する．フィールドは，equals メソッドもしくは hashCode メソッドで参照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 型で表す．これは，equals メソッドにおけるプリミティブ型は等価判定さえできればモデルに含める情報量としては十分であり，hashCode メソッドにおけるプリミティブ型はラッパーメソッド等を使用して int 型に変換しているからである．参照型のフィールドは参照型を表すレコード Ref 型を定義し，それを用いて表現する．Ref 型には，等価判定で使われる値とハッシュコード値，そのオブジェクトのポインタの 3 つの int 型の値を持つ．Ref 型オブジェクトは，手法の入力として与えられた型階層外のクラスに対して用いられ，equals メソッドと hashCode メソッドを規則通りに実装していることを前提にしている．配列や List，Set，Map のフィールドは SMT-LIB の配列やリストを用いて表現する．ただし，後の検査のためにサイズは 10 で固定する．

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．また，Java における instanceof 演算子を表す関数を定義する．Object クラスの定義では型階層中の全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，int 型のポインタと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表す値を持つ．

提案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 this，that，other という名前の 3 つの Object 型の変数を宣言する．また,「null でない任意の参照値 x について，x.equals(null) は false を返す」という規則の検査のために，nobj という名前の，ポインタの値が 0 である Object 型の変数も宣言する．

3.4.2 メソッドの変換

Java におけるメソッドを SMT-LIB でどのように表すかについて述べる．まず，Jimple の各文に対して式木を作成する．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，基本的に各文は 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 4 つで構成されている．この情報を用いて変数の参照の連鎖を 2 分木で表したものが式木である．変換は式木の葉から根に向けて，再帰的に変換ルールを適用することで行う．単純な変換が可能な演算に対する変換ルールを表 2 に示す．Java コードから SMT-LIB への変換関数を μ とし，以下では boolean 型，int 型をそれぞれ 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．

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 へと変

換する．型のチェック，状態のチェックは，表 2 の通りに変換する．配列，List，Set，Map の等価判定については，格納されている要素同士の比較を表 2 の通りに変換する．

hashCod メソッドの変換では基本的に return 文を根とする式木を表 2 の通りに変換するが，パターンの解析で得られた 3 つのパターンについては特別に変換する．まず，int 型への変換されている変数は最初から int 型であるかのように扱う．例えば，float 型のフィールドを int 型へ変換している場合は，そのフィールドは最初から int 型としてモデル化する．ビット演算は，その被演算子をビットベクトル型の変数で表し，表 2 の通りに変換する．そして，演算の結果を bv2int 関数を用いて int 型へ変換する．Java における int 型は 32 ビットであるが，32 ビットでモデル化を行うと膨大な検査時間を要するので 8 ビットとする．ループによる算術演算では，ループ中の式がループ回数によって最終的な式が決定するのに対して SMT ソルバに動的に式を評価する仕組みがないため，ループを扱えない．そこで，ループ

表 2: 単純な変換が可能な演算

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

回数が 0 回の場合，1 回の場合，2 回の場合，… という具合にループ回数の違いによるモデル化を行い，各モデルについて検査を行う．3.4.1 節で述べたように，配列などのサイズは 10 で固定しているため，10 回の場合まで検査を行う．

3.5 SMT ソルバによる検査

SMT ソルバによる検査では Z3 を用いて SMT-LIB で記述されたモデルを検査する．検査項目は equals メソッドが満たさなければならない規則の全てと，hashCode メソッドが満たさなければならない規則のうち 1 つである．前述したように，フィールドのサブセット判定は equals メソッド，hashCode メソッドの処理パターンの解析と平行して行うため，ここでは検査しない．SMT-LIB における assert 制約式を用いて，各規則を表した条件式の否定がそれぞれ充足可能か調べる．ある制約式が充足可能な場合，手法の入力として与えられた型階層はその制約式に対応する規則に違反している．具体的にどのクラスがどういった変数の値の時にその規則を満たさないのかは，制約式の充足例として表示される．

4　実装

本章では，3 章で述べた手法を実装したツールについて述べる．

4.1　実装概要

ツールは Java を用いて Eclipse プラグインとして実装した．ツールは Eclipse 上の Java プロジェクトを入力とし，フィールドのサブセット判定を Eclipse のビューに出力し，他の規則の検査結果はテキストファイルで出力する．ツールは GUI となっておりマウスを用いた簡単な操作で検査を行うことができる．ツールの実装規模は空行やコメントを除いて 25,571 行となった．

4.2　ツールの詳細

ツールは既存研究の手法が実装されている EQ と呼ばれるツール [8] を拡張して実装した．

EQ には Eclipse 上の Java プロジェクトから equals メソッドに関する型階層を構築する処理があるので，equals メソッドだけではなく hashCode メソッドにのみ関係しているクラスも含むような型階層を構築するように拡張した．この処理は JDT を用いた 3 つのステップからなる．まず，Java プロジェクトの中から Java コードを探し出す．次に，Java コードから equals メソッドもしくは hashCode メソッドを持つクラスを発見する．そして，そのクラスの継承関係を元に型階層を構築する．また，パス解析やメソッドの処理パターンの解析の部分も EQ では equals メソッドのみを解析しているが，hashCode メソッドも解析するように拡張した．EQ における Alloy への変換と Alloy Analyzer による検査は，ツールではそれぞれ SMT-LIB への変換と SMT ソルバによる検査を行うようにした．

また，ツールではマルチスレッドプログラミングを行うことで処理の高速化を図っている．例を挙げると，ツールでは 1 つの型階層に対して 1 つのスレッドを用いてパス解析，メソッドの処理パターンの解析，SMT-LIB への変換を一貫して行う．スレッドは，JVM が使用可能なプロセッサの数まで同時に作成できる．型階層がプロセッサの数より多い場合は，型階層の待ち行列を組み処理を終えたスレッドへと適宜型階層を割り当てていく．

なお，配列や List，Set，Map に対する変換およびビット演算の変換は本ツールにおいて未実装である．そのため，型階層内でそれらのパターンが 1 つでも用いられていた場合，SMT-LIB への変換は行わず，SMT ソルバによる検査は行わない．ただし，フィールドのサブセット判定は可能であるため実行する．

4.3 使用方法

まず，図 6 の赤い虫のボタンをクリックする．すると，図 7 のようなダイアログが表示されるので検査したい Java プロジェクトを選択し，「OK」をクリックする．プロジェクトの検査が始まると図 8 のようなプログレスバーが表示されるので，検査が終了するまでしばらく待つ．検査が終了すると「Analysis completed successfully!」のポップアップメッセージが表示され，図 9 のようにクラス階層と検査結果の一部がビューに表示される．また，残りの検査結果もテキストファイルとして対象プロジェクトのディレクトリ下に出力される．

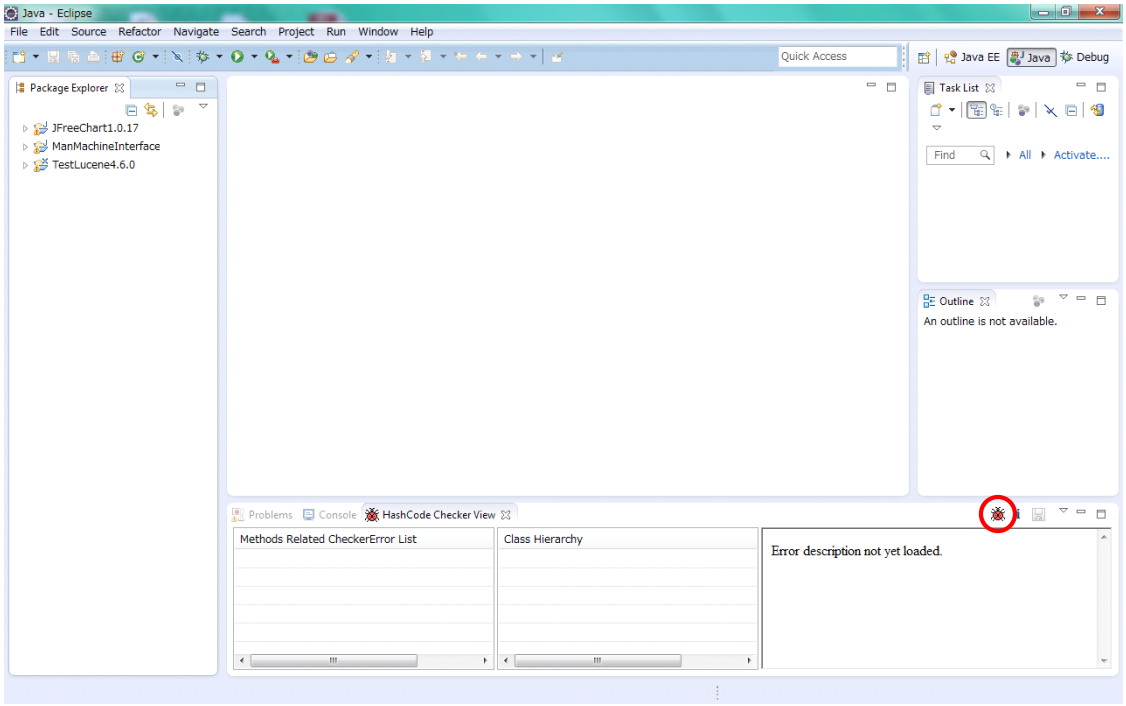

図 6: 起動画面

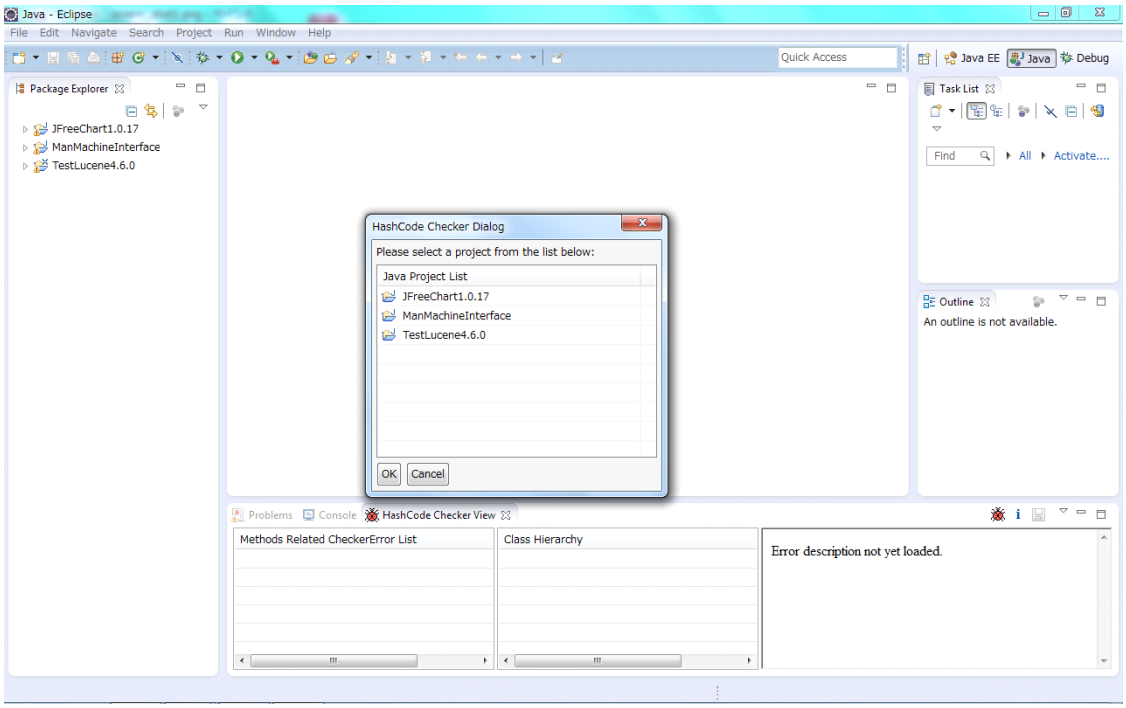

図 7: 対象プロジェクト選択画面

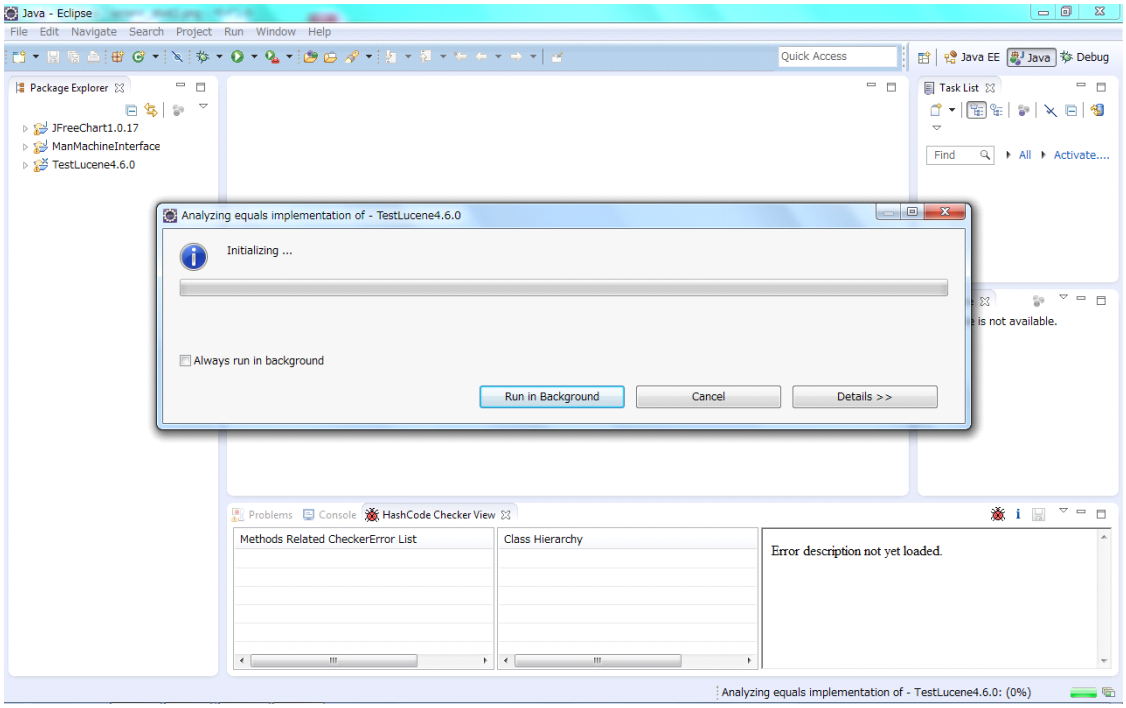

図 8: 検査実行中画面

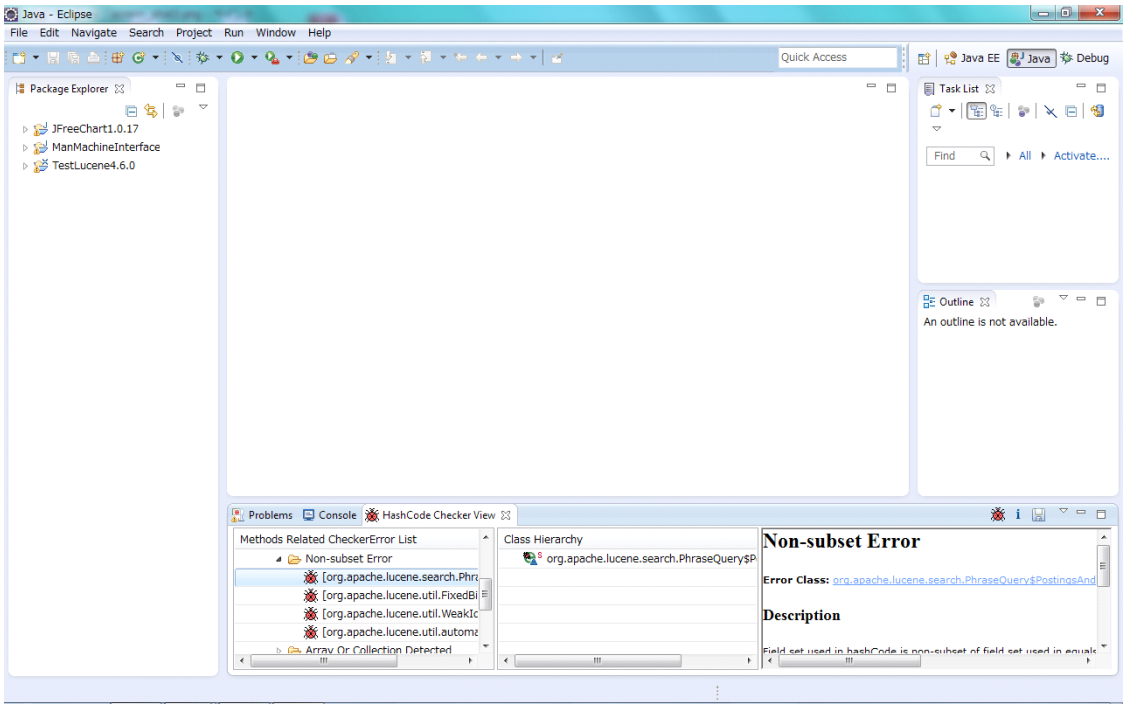

図 9: 検査結果表示画面

5　評価実験

本章では，実装したツールを用いて行った評価実験について述べる．

5.1　実験概要

本報告では実験 1 と実験 2 の 2 つの評価実験を行った．実験 1 では，実プロジェクトにおいて規則に違反している実装がどの程度存在するか調査することを目的として，ツールを用いて実プロジェクトを検査した結果と検出された規則違反の考察を行った．実験 2 では，提案手法のスケーラビリティの評価を目的として，規模の異なるプロジェクトに対するツールの実行時間の比較を行った．次節以降でそれぞれの実験の詳細について述べる．

5.2　実験 1

Lucene4.6.0[32]，Tomcat8.0.1[33]，JFreeChart1.0.17[34] の 3 つのオープンソースプロジェクトに対して，ツールを用いて検査を行った．これにより，実プロジェクトにおいて規則に違反している実装がどの程度存在するか調査した．

5.2.1　結果と考察

表 3 に実験結果を示す．各規則名の列はその規則に違反している型階層の数を表している．

ツールを用いて検出された規則違反についての考察を述べる．equal メソッドの規則に違反している原因としては文献 [9] で述べられている非対称 null チェック，型階層における不適当な型チェック，タイピングミスの 3 つが確認された．また，フィールドに対するメソッド呼出しを非決定性関数としてモデル化したため正しくモデル化できていないことが原因となって規則全てに違反しているものが 3 つ確認された．このような原因に対しては既存研究で行われているように，展開されないメソッドの振舞いに関する情報をユーザが入力できるようにツールを拡張することで改善されると考えられる．

表 3: オープンソースプロジェクトに対するツールの適用結果

プロジェクト名	equls 規則違反				hashCode 規則違反		合計
	反射性	対称性	推移性	null	サブセット規則	等価規則	
Lucene	2	0	0	0	4	1	7
Tomcat	11	3	4	3	14	7	35
JFreeChart	1	1	2	0	76	36	113

hashCode メソッドの規則のうちサブセット規則に関しては，文献 [10] で述べられているように，一度計算したハッシュコードの値などを equals メソッドで参照されないフィールドに保持しておき，hashCode メソッドでそのフィールドを参照していることが原因となって規則に違反してものが見られた．このような違反は本来の hashCode メソッドの規則に完全に違反しているとは言い切れない．また，等価規則に関しては equals メソッドがオーバーライドされているクラスにおいて hashCode メソッドがオーバーライドされていないために規則に違反しているものが多数見られた．このような違反は JFreeChart のみに見られ他の 2 つのプロジェクトには見られなかったので，プロジェクト全体の実装方針によって違反数に大きな差が出たのではないかと考えられる．前述したように equals メソッドをオーバーライドする場合は hashCode メソッドも同時にオーバーライドすべきであるため，プロジェクトの方針として hashCode メソッドをオーバーライドするように定めべきである．また，equals メソッドが規則に違反しているため等価規則を満たすことが不可能なものが 2 つ確認された．

5.3 実験 2

Lucene4.6.0，Tomcat8.0.1，JFreeChart1.0.17 の 3 つのプロジェクトに対してツールを適用して，実行時間の比較を行った．これにより，提案手法のスケーラビリティの評価を行った．

5.3.1 結果と考察

表 4 に実験結果を示す．総パス長の列はプロジェクトに含まれるパスの長さの合計を，各ステップ名の列は各ステップの処理時間を表している．

実験結果から，実行時間が非常に長い JFreeChart において総パス長が大きいことが分かった．よって，提案手法は大規模なプロジェクトより小中規模なプロジェクトに対して有用であると言える．ただし，プロジェクト中において検査する型階層を選択できるようにすることで実行時間の短縮を図ることができるため，大規模プロジェクトへの適用も十分可能であると考えられる．また，メソッドの処理パターンの解析・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ことが分かる．よって，

表 4: 実行時間の比較

プロジェクト名	総パス長	型階層構築	パス解析	メソッドの処理パターン解析・変換	SMT ソルバによる検査	実行時間
Lucene	16,970	5s	12s	29s	1s	48s
Tomcat	257,590	3s	38s	4m	2s	4m45s
JFreeChart	3,538,281	8s	3h6m21s	3h11m31s	6s	6h18m9s

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図れると考えられる.

6　あとがき

本研究では，著者の属する研究グループで提案された 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の実装と評価を行った．

提案手法の実装として，Eclipse のプラグインとしてツールを作成した．これは，Eclipse 上のプロジェクトを入力とし 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出力するものである．

実装したツールをいくつかのオープンソースプロジェクトへ適用し，実プロジェクトにおいて規則に違反している実装がどの程度存在するかの調査と提案手法のスケーラビリティの評価を行った．結果として，規則に違反している実装例をいくつか発見することができ，さらに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ことが分かった．大規模なプロジェクトに対しても，適用範囲を選択することで現実的な処理時間で検査を行うことができると考えられる．

今後の課題として，ツールにおいて未実装の機能である配列や List，Set，Map，ビット演算を含むメソッドへの対応，プロジェクト中において検査する型階層を選択できるようにする機能の実装などが挙げられる．

謝辞

本報告を行うにあたり，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 教授に心より感謝申し上げます.

本報告の全過程を通し，終始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.

本報告に関して，的確なご助言ご指導を頂きました 井垣 宏 特任准教授に心より感謝申し上げます.

本報告を行うにあたり，日常の議論の中でご助言を頂きました 肥後 芳樹 助教に心より感謝申し上げます.

本報告を行うにあたり全面的なご指導，ご協力を頂きました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博士前期課程 2 年の榛葉 浩章 氏，同 1 年の大田 崇史 氏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協力に心より感謝致します.

最後に，本研究に至るまでに，講義，演習，実験等でお世話になり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] Bloch, J.: *"Effective Java"*, Addison-Wesley (2008).

[2] Oracle: "Java Platform, Standard Edition 7 API Specification" (2013). http://docs.oracle.com/javase/7/docs/api/.

[3] Hovemeyer, D. and Pugh, W.: "Finding bugs is easy", *ACM SIGPLAN Notices Homepage archive*, pp. 92–106 (2004).

[4] Dolby, J., Vaziri, M. and Tip, F.: "Finding bugs efficiently with a SAT solver", *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] Vaziri, M., Tip, F., Fink, S. and Dolby, J.: "Declarative Object Identity Using Relation Types", *Proceedings of the 21st European Conference on Object-Oriented Programming*, pp. 54–78 (2007).

[6] Rupakheti, C. R. and Hou, D.: "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).

[7] Rupakheti, C. R. and Hou, D.: "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).

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

[9] Rupakheti, C. R. and Hou, D.: "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).

[10] 榛葉浩章 "Java における equals メソッドと hashCode メソッドの整合性の検査"，修士論文，大阪大学 (2013).

[11] Lam, P., Bodden, E., Lhoták, O. and Hendren, L.: “The Soot framework for Java program analysis: a retrospective”, *Proceedings of the Cetus Users and Compiler Infrastructure Workshop* (2011).

[12] Jackson, D.: “Alloy: a lightweight object modelling notation”, *ACM Transactions on Software Engineering and Methodology*, Vol. 11, No. 2, pp. 256–290 (2002).

[13] Jackson, D.: *“Software abstractions: logic, language, and analysis”*, The MIT Press (2011).

[14] Jackson, D.: “ Alloy: a language and tool for relational models” (2013). http://alloy.mit.edu.

[15] 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).

[16] Dutertre, B. and de Moura, L.: “A Fast Linear-Arithmetic Solver for DPLL(T)”, *Proceedings of the 18th international conference on Computer Aided Verification*, pp. 81–94 (2006).

[17] Barrett, C. and Tinelli, C.: “CVC3”, *Proceedings of the 19th International Conference on Computer Aided Verification*, pp. 298–302 (2007).

[18] Griggio, A.: “A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic”, *JSAT*, Vol. 8, pp. 1–27 (2012).

[19] Cimatti, A., Griggio, A., Schaafsma, B. J. and Sebastiani, R.: “ The MathSAT5 SMT Solver”, *Proceedings of the 19th international conference on Tools and algorithms for the construction and analysis of systems*, pp. 93–107 (2013).

[20] de Moura, L. and Bjorner, N.: “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).

[21] Cok, D., Stump, A. and Deters, M.: “SMT-COMP2012” (2012). http://smtcomp.sourceforge.net/2012/.

[22] Barrett, C., Stump, A. and Tinelli, C.: “The SMT-LIB Standard Version 2.0” (2012). http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf.

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

[24] Rayside, D., Benjamin, Z., Singh, R., Near, J. P., Milicevic, A. and Jackson, D.: "Equality and Hashing for (almost) Free: Generating Implementations from Abstraction Functions", *Proceedings of the 31st International Conference on Software Engineering,*, pp. 342–352 (2009).

[25] Grech, N., Rathke, J. and Fischer, B.: "JEqualityGen: Generating Equality and Hashing Methods", *Proceedings of the ninth international conference on Generative programming and component engineering*, pp. 177–186 (2010).

[26] Jensen, T., Kirchner, F., Pichardie, D. and Atlantique, I. R. B.: "Secure the clones: Static enforcement of policies for secure object copying", Technical report (2010).

[27] Anastasakis, K., Bordbar, B., Georg, G. and Ray, I.: "UML2Alloy: A Challenging Model Transformation", *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, pp. 436–450 (2007).

[28] Liu, T., Nagel, M. and Taghdiri, M.: "Bounded Program Verification using an SMT Solver: A Case Study", *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pp. 101–110 (2012).

[29] Gent, I. P., Jefferson, C. and Miguel, I.: "Minion: A Fast, Scalable, Constraint Solver", *Proceedings of the 17th European Conference on Artificial Intelligence*, pp. 98–102 (2006).

[30] Balasubramaniam, D., Jefferson, C., Kotthoff, L., Miguel, I. and Nightingale, P.: "An Automated Approach to Generating Efficient Constraint Solvers", *Proceedings of the 2012 International Conference on Software Engineering*, pp. 661–671 (2012).

[31] Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M. and Poll, E.: "An overview of JML tools and applications", *International Journal on Software Tools for Technology Transfer*, pp. 212–232 (2005).

[32] Apache: "Apache Lucene - Welcome to Apache Lucene" (2012). http://lucene.apache.org/.

[33] Apache: "Apache Tomcat - Welcome" (2014). http://tomcat.apache.org/.

[34] Limited, O. R.: “JFreeChart” (2012). http://www.jfree.org/jfreechart/.