CLARKSON UNIVERSITY

A Path-based Approach For Analyzing Object Equality in Java

A Thesis by

Chandan R. Rupakheti

Department of Mathematics and Computer Science

Submitted in partial fulfillment of the requirements

for the degree of

Master of Science

Computer Science

May 2010

Accepted by the Graduate School

Date DEAN

The undersigned have examined the thesis entitled **A Path-based Approach For Analyzing Object Equality in Java** presented by Chandan R. Rupakheti, a candidate for the degree of Master of Science, Computer Science and hereby certify that it is worthy of acceptance.

_____

Date

EXAMINING COMMITTEE

_____

Christopher Lynch

_____

Jeanna Matthews

_____

ADVISOR Daqing Hou

Abstract

Applications built on top of an existing design and implementation are in general expected to collaborate well with that design and respect all of its intent. Failure in achieving this may result in buggy, fragile, and less maintainable code in the applications. When the dependence on an existing design becomes more widespread, this requirement on proper extension obviously becomes even more critical. As an instance of this general problem, the design for object equality in Java and its extensions are examined in detail and empirically. By examining how object equality is extended in a large amount of Java code, a set of typical problems are detected through a checker and their root causes analyzed. A set of design guidelines for object equality is proposed, which, if followed, will help programers systematically design and evolve rather than hack a solution. Examples are drawn from a case study of multiple industrial and open source projects to illustrate the identified problems and how the proposed guidelines can help solve these problems. Furthermore, a complete set of unsupportable problems are also discussed to show the limitation of the techniques and the complexity in the implementation of the checker.

Acknowledgements

I am very thankful to Dr. Daqing Hou, Dr. Christopher Lynch, Dr. Jeanna Matthews and Dr. Francois Bronsard for their guidance and constructive comments on the thesis. Thanks to my mother Chanda Sharma and my father Chandrika P. Rupakheti for their love and support. I am also grateful to my brother Chetan Raj Rupakheti, my friend Manila Poudyal and folks at Software Engineering Research Laboratory (SERL) for their help.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Equivalence Relation in Java

It is common for object-oriented programming languages like Java and C# to provide a useful set of collection data types like set, map, vector, and hash table [42, 33] [1]. In order to work as elements inside such a container data structure, application objects need to support an equality predicate with which a pair of objects can be compared. The design adopted by both Java and C# is to specify a contract for equality in the *Object* class, which is expected to be supported by all other application classes. In this way, the collection data types can be implemented with the assumption that the element objects will honor this equality contract.

In particular, in Java, the *equals(Object)* method in the *java.lang.Object* class is designed to support the work of collection types. As its behavioral specification, the *equals* method is required to implement an *equivalence relation* on any two *non-null* object references x and y, supporting the following properties:

1. **Reflexivity**: *x.equals(x)* always returns *true*.

2. **Symmetry**: *x.equals(y)* returns *true* if and only if *y.equals(x)* returns *true*.

[1] This chapter in-part has been published in [37].

3. **Transitivity**: for any non-null reference *z*, if *x.equals(y)* and *y.equals(z)* return *true*, then *x.equals(z)* should return *true*.

4. **Consistency**: multiple invocations of *x.equals(y)* consistently return true or false, provided that no information used in comparing the objects is modified.

5. **Non-nullity** [2]: *x.equals(null)* must return *false*.

All objects that become an element of a collection from the *Java Collection Framework* [9] must obey this contract in order to function properly inside the collection.

```
1  public class Point3D extends Point2D {
2     private int z; // the z coordinate
3     public boolean equals(Object o) {
5         if ((o instanceof Point3D)) {
6           return super.equals(o) &&
7              (Point3D)o.z==this.z;
8         }
9         else return super.equals(o);
10   }
11 }
```

Figure 1.1: An implementation of Point3D and equals() that violates transitivity (pp. 182 of [30], with modification of class names).

This issue of object equality is a general design problem present in both Java and C#. It is also a long-standing problem that can be traced back to the Lisp community [7]. It has a wide impact on many classes. For example, 624 classes in JDK 1.5 implement an *equals()* method, covering areas such as security, corba implementation, utility classes, collection types, GUI, and so on. However, implementing *equals()* to respect this contract needs several considerations that can be easily overlooked by a developer, resulting in buggy or fragile code. Such defects are notoriously difficult to find, even for experienced programmers. To make matters worse, textbooks often present flawed versions of these critical operations (e.g., the example shown in Figure 1.1) or give unsound advices. The understanding of the problem is so vague that even popular Java IDEs like Eclipse 3.5 [17] and NetBeans 6.8 [35] have bugs in the *equals()* method generated through their wizard. It is also the center of much controversy. For example, type-incompatible equality [3] between a supertype

[2]The term non-nullity is due to Bloch [10].

[3]Type-incompatible equality will be defined in Section 1.3.

and a subtype would violate the well-known Liskov Substitution Principle [30, 31]. Should type-incompatible equality be therefore prohibited? Furthermore, should mutable types be allowed to define *equals()*?

1.2 Contribution

The thesis provides a broad insight on the process of designing, implementing and validating an *equals* method. The contributions of this work can be summarized as follows:

Types of Equality We identify three different kind of equality: Type Compatible, Type Incompatible and Hybrid Equality and provide reasons for such separation with examples. We also show how *equals* method implementation becomes error prone under the lack of clear understanding of these equalities.

Variation in Equality Implementation By investigating several open source projects including JDK 1.5, we show different variations and techniques for implementing the *equals* method. Furthermore, we expose problems associated with several bad implementations that can help the readers write a better *equals*.

Design guidelines We provide stepwise guidelines for designing and implementing the *equals* method in a type hierarchy. These guidelines, if followed properly, will help reduce problems associated with *equals* implementation.

Static Checker We develop a path-based static checker for identifying problems with *equals* implementations. Path-based analysis has been avoided due to path explosion problems in a whole program analysis. We show how a *type hierarchy based program slicing* technique and *filter as-you-go* strategy significantly reduce unnecessary paths while still providing a good static approximation of runtime behavior of the code. We also show how a path-based analysis can be used for a Java to Alloy [26, 27] code translation for model checking. This is done by first translating Java to Jimple [44] (A 3-address intermediated representation) from which control flow paths are extracted

and converted to Alloy module.

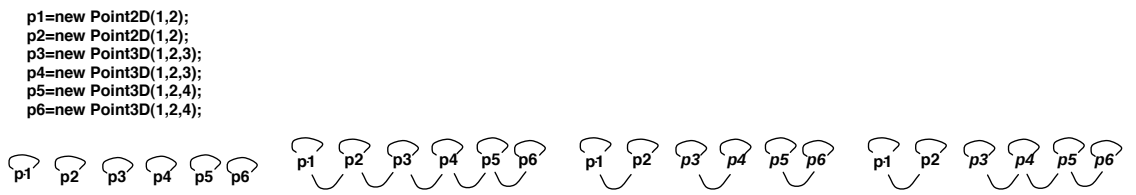
Pattern Detection and Alloy Model We present several pattern detection algorithm for *equals* code patterns that can be abstracted to a much higher level logical construct in Alloy. While doing so, we develop models for different Java-based data structures in Alloy. In particular, we show abstraction techniques for Java Arrays, Lists, Sets and Maps data structures. Furthermore, the multiple inheritance model of Java in Alloy is also discussed.

False Positives and Checker Limitations We validate our checker by applying it to several open source projects. We discuss reported problems, false positives and most importantly, we categorize different instances of *equals* implementation that are not handled by the checker or could not be translated to the Alloy model for model checking. This will not only help identify the limitation of our approach, but also provides insight into future research directions.

The thesis is organized as follows: In Chapter 1, we distinguish three kinds of equality and elaborate on their implementations. A preliminary case study of the equality design is performed with the help of four open source projects in Chapter 2. Findings from the case study are then used for building a path-based static checker for analyzing *equals* implementations in Chapter 3. A full-scale validation of the checker is done in Chapter 4 with categorical reporting of detected problems and false positives. In, Chapter 5 we present related work and finally in Chapter 6 we present future work and conclude the thesis.

1.3 Design Intent and Implementation Patterns for Equality

In this section, the equality relation is analyzed and three kinds of equality (type-compatible, type-incompatible, and hybrid equality) are introduced. Their relation with a type hierarchy and how they can be designed and implemented properly in a type hierarchy is described. In particular, the relation between type-incompatible equality and subtyping is discussed.



(a) Reference equality (b) Type-compatible equality defined in terms of x, y, or both (c) Type-incompatible (d) Type-incompatible equality

Figure 1.2: Examples of equality relations for Point2D and Point3D. For Point3D, the equality in 1.2c is defined in terms of x, y, and z, and the one in 1.2d is defined in terms of x and y. The links between two nodes represent equality. Links that can be inferred via transitivity are omitted for clarity.

1.3.1 Type, state, and equality relation

A type is defined by a value space (a.k.a state) of named properties as well as operations on the value space. For example, an object of type *Point2D* may have two properties, the x and y cartesian coordinates, and setters and getters for these properties. Two types may be related by subtyping, which is governed by the Liskov Substitution Principle (LSP hereafter) such that an object of a subtype can be substituted for an object of a supertype in all contexts where a supertype object is expected [30, 31]. For example, if it is intended and is legitimate for a *Point3D* object to appear in all of the contexts where a *Point2D* is used, then *Point3D* can be made a subtype of *Point2D*. Note that the appropriateness of subtyping is often determined by domain semantics and contexts of use. Finally, each object has a reference (address) that uniquely identifies the object, and each object has one or more types.

Given a set of objects, multiple equivalence relations can be defined. In theory, given a set of objects that share n properties, 2^n equivalence relations can be defined in terms of these properties. Figure 1.2 shows several examples of equality defined for objects of *Point2D* and *Point3D*.

There are two special cases of equality. One is *reference equality* [4] where an object is uniquely identified by its reference, and thus can be equal only to itself. Therefore, ref-

[4]Bloch uses identity equality for what we call reference equality. [10]

```
1  public class Point2D {
2  ... defining representation & operations
3  public boolean equals(Object o) {
4         if (!(o instanceof Point2D))
               return false;
5         Point2D that = (Point2D)o;
6         return this.getX() == that.getX() &&
7               this.getY() == that.getY();
8  }
9 }
```

Figure 1.3: Implementing type-compatible equality with *instanceof*.

erence equality is the most discriminating equality relation. The default implementation in *Object.equals()* is a reference equality. When reference equality is too discriminating to be useful, the default implementation can be overridden by another notion of equality based on domain semantics. The other special case of equality, which we call *full equality*, takes into account the full value space of objects. *Full equality* is a special case of domain-semantics-based equality and is suitable for testing whether two objects are fully behaviorally equivalent rather than just partially similar.

1.3.2 Type-compatible and type-incompatible equality

As the first step in designing an equality relation, it is always helpful to distinguish between the following two intents with respect to a notion of compatibility between object types. *Type-compatible equality* is determined by comparing a subset of common properties between two types S and T, which may or may not be in the same type hierarchy. *Type-incompatible equality* requires that only objects from the same type can possibly be considered equal, and the equality between any pair of objects from the same type can be further determined by comparing a subset of the properties between the two objects.

If two types involved in a type-compatible equality form a type hierarchy, the type hierarchy can help reduce the number of *equals* needed to be implemented due to inheritance and dynamic dispatching [30]. Figure 1.3 illustrates how an equality between *Point2D* (superclass) and *Point3D* (subclass) can be implemented by *equals()* in class *Point2D* and inherited by *Point3D*. The type testing at line 4 checks whether o can be type-cast to *Point2D*. *o instanceof Point2D* will return true if o's type is either *Point2D* or one of its

```
1  public class Point2D {
2  ... defining representation & operations
3  public boolean equals(Object o) {
4     if (o==null) return false;
5     try { Point2D that = (Point2D)o;
6         return this.getX()==that.getX()&&
7              this.getY() == that.getY();
8     } catch (ClassCastException cce){
9           return false;
10   }
11  }
12 }
```

Figure 1.4: Implementing type-compatible equality with exception handling rather than *instanceof*.

```
1  public class Point2D {
2  ... defining representation & operations
3  public boolean equals(Object o) {
4     if (o == null) return false;
5     if (!o.getClass().equals(this.getClass())
          return false;
6     Point2D that = (Point2D)o;
7     return this.getX() == that.getX() &&
8            this.getY() == that.getY();
9    }
10 }
11  public class Point3D extends Point2D{
12  ... defining representation & operations
13  public boolean equals(Object o) {
14     if (o == null) return false;
15     if (!o.getClass().equals(this.getClass())
            return false;
16     Point3D that = (Point3D)o;
17     return this.getX() == that.getX() &&
18            this.getY() == that.getY() &&
19            this.getZ()==that.getZ();
20  }
     }
```

Figure 1.5: Implementing type-incompatible equality with *getClass()*.

subtypes, and it will return *false* when *o* is *null*. Line 5 does the type cast, and lines 6 and 7 compare the state.

The *instanceof* type testing can also be done with the exception handling mechanism. Figure 1.4 shows an implementation that is functionally equivalent to the one in Figure 1.3. While using exception handling may result in a small benefit in performance, it is not a common way of implementing *equals()*. It also increases the number of implementation patterns a developer has to master and understand.

Figure 1.5 shows an implementation for a type-incompatible equality between *Point2D* and *Poin3D*. This implementation uses the Java reflection API *getClass()*, which returns the runtime type of the receiver object. Therefore, the tests on lines 5 and 15 will permit

only objects of the same type to pass. The null tests at lines 4 and 14 address the *non-nullity* property and ensure the subsequent calls to *getClass()* will not throw null pointer exceptions. When the equality is defined in terms of the same set of properties shared by two types, subclasses may inherit *equals()* from the superclass rather than implement its own. This is impossible for the example in Figure 1.5 because *equals()* in *Point3D* adds the z dimension.

When two types involved in a type-incompatible equality are intended to form a subtyping relation but the two *equals()* are implemented in terms of their respective type and state, the pair of *equals()* will be incompatible under LSP and will cause the two types to not form a proper subtyping relation. (That is, the specification of *Point2D.equals()* states that *Point2D* can only be equal to another *Point2D* that has identical x and y, and *Point3D.equals()* states that *Point3D* can only be equal to another *Point3D* with identical x, y, and z.) In this case, *equals()* should be excluded from the type specifications, and the rest of the operations in the two types can still conform to LSP. It can then be required that such *equals()* must not be used in contexts where objects of a subtype are substituted for those of a supertype. But these two *equals()* do conform to LSP with *Object.equals()* because the specification of *Object.equals()* is weaker than the afore-mentioned ones with respect to how equality is exactly defined.

1.3.3 Hybrid equality

Sometimes, it can be useful or even necessary to define an equality that mixes type-compatible and type-incompatible equality in the same hierarchy (*hybrid equality*). When the main hierarchy is type-incompatible and a sub-hierarchy is type-compatible, they can be implemented respectively using the techniques introduced above. However, when a type-compatible hierarchy contains a type-incompatible sub-hierarchy, a new implementation pattern is needed. In what follows, it is first shown that it is impossible to use the techniques introduced so far to implement this kind of hybrid equality in a way that satisfies the equals contract. An implementation for hybrid equality based on the template method

pattern [22] is then introduced.

```
public boolean equals(Object o) {
    if (!(o instanceof ColorPoint)) return false;
        ColorPoint that = (ColorPoint)o;
        return this.getX()==that.getX() &&
            this.getY()==that.getY() &&
            this.getColor() == that.getColor();
}
```

Figure 1.6: Color sensitive equals for ColorPoint.

Suppose a type-compatible equality has been implemented for *Point2D* and *Point3D* as shown in Figure 1.3. Now a new subclass *ColorPoint* needs to be added to *Point2D*, and suppose *ColorPoint* needs to implement a type-incompatible equality with regards to *Point2D* and *Point3D*. Figure 1.6 shows an implementation that attempts to provide type-incompatible equality at the *ColorPoint* side, but loses the symmetry property. The loss of symmetry can be demonstrated by the following code snippet:

```
Point2D p1 = new Point2D(1,2);
ColorPoint p2 = new ColorPoint(1,2,Color.RED);
p1.equals(p2); // returns true
p2.equals(p1); // returns false (broken symmetry)
```

An attempt to fix the symmetry violation but at the expense of the transitivity property is shown in Figure 1.7. The loss of transitivity can be illustrated by the following code snippet:

```
ColorPoint p1 = new ColorPoint(1,2,Color.RED);
Point2D p2 = new Point(1,2);
ColorPoint p3 = new ColorPoint(1,2,Color.BLUE);
p1.equals(p2); // returns true
p2.equals(p3); // returns true
p1.equals(p3); // false (broken transitivity)
```

```
public boolean equals(Object o) {
    if (!(o instanceof Point2D)) return false;
    // For Point2D, ignore color in comparison
    if (!(o instanceof ColorPoint))
        return o.equals(this);
    // o is a ColorPoint; compare color as well
    ColorPoint that = (ColorPoint)o;
    return this.getX()==that.getX() &&
            this.getY()==that.getY() &&
            this.getColor() == that.getColor();
}
```

Figure 1.7: Color insensitive equals for ColorPoint.

The root cause of this problem on transitivity is that the two equality comparisons that *ColorPoint* participates in make use of different properties. When compared with a *Point2D*, color is ignored. When compared with a *ColorPoint*, color is included.

The loss of symmetry in the implementation of Figure 1.6 is caused by the fact that *equals()* of *Point2D* fails to include the color property of *ColorPoint*. When the argument is a *ColorPoint*, this implementation will compare only the x and y coordinates but not the color, resulting in a violation of symmetry. This violation can be fixed by giving the argument an opportunity to add its own properties in addition to those of the superclass.

Figure 1.8 shows how *Point2D* and *ColorPoint* can be modified to achieve this. Notice that a new method *equalsDelegate()* is introduced and called by *equals()* in *Point2D*. *ColorPoint* overrides *equalsDelegate()* to add its type-specific comparison. In this case, *ColorPoint* enforces the requirement that the argument must also be a *ColorPoint*. It can be verified that the new implementation satisfies the equals contract. Also notice that the *equals()* method is declared final so that no subtypes of *Point2D* can override it. An incompatible subtype such as *ColorPoint* needs to override *equalsDelegate()* method to perform type-specific comparison. Compatible subtypes will simply inherit both *equals()* and *equalsDelegate()*.

```
public class Point2D {
  public final boolean equals(Object o) {
    if (!(o instanceof Point2D)) return false;
    Point2D that = (Point2D)o;
    return comparing x and y &&
    // For symmetry not provided by instanceof
    that.equalsDelegate(this)&&equalsDelegate(o);
    }
 // type-specific comparison. true by default.
 protected boolean equalsDelegate(Object o) {
    return true;  }
 // Remainder omitted
}

public class ColorPoint extends Point2D {
  protected boolean equalsDelegate(Object o) {
    if (!(o instanceof ColorPoint)) return false;
    ColorPoint that = (ColorPoint)o;
    // return the comparison of x, y, and color;
    }
 // Remainder omitted
}
```

Figure 1.8: Implementation of hybrid equality.

```
1  public class Point2D {
2  ... defining representation & operations
3  public boolean equals(Object o) {
4     if (o == null) return false;
5     if (o.getClass().equals(Poin2D.class){
6     Point2D that = (Point2D)o;
7     return this.getX() == that.getX() &&
8              this.getY() == that.getY();
9      }
10     if (o.getClass().equals(Poin3D.class){
11     Point3D that = (Point3D)o;
12     return this.getX() == that.getX() &&
13              this.getY() == that.getY();
14      }
15   return false; }
16 }
1  public class Point3D {
2  ... defining representation & operations
3  public boolean equals(Object o) {
4     if (o == null) return false;
5     if (o.getClass().equals(Poin2D.class){
6       // return result of comparison
7      }
8     if (o.getClass().equals(Poin3D.class){
9     Point3D that = (Point3D)o;
10     // return result of comparison
11    }
12    return false;}
13 }
```

Figure 1.9: Implementing type-compatible equality between types not in the same hierarchy.

By calling *equalsDelegate()* twice (*that.equalsDelegate(this)* && *equalsDelegate(o)*), *equals()* of *Point2D* in Figure 1.8 essentially provides a bi-directional check for the two types involved in a comparison. As discussed above, the first check (*that.equalsDelegate(this)* is to give a subtype an opportunity to do a subtype-specific comparison. The purpose of the second check becomes evident when considering the evaluation of *aColorPoint.equals(aPoint2D)*, where *equalsDelegate(o)* will cause *equalsDelegate()* of *ColorPoint* to be called, which will correctly return *false*.

It can be verified that all the implementations (Figures 1.3, 1.4, 1.5, and, 1.8) satisfy the equals contract introduced in Section 1.1.

1.4 Discussion

It is even possible to define an equivalence relation where two objects from two different type hierarchies are considered equal. Such equality may be useful in scenarios where both kinds of objects are used as a key to a hash table. Two *equals()* need to be implemented, one

for each type. Figure 1.9 shows how this can be done under the assumption that *Point2D* and *Point3D* are independent classes. However, in the projects we studied, no instances of equality were found being done in this way. This is probably because when two types share properties, a type hierarchy can usually be designed to relate them. In this thesis, we focus on the case where any two objects that are to be considered equivalent also belong to the same type hierarchy and on how to design and implement *equals()* in a type hierarchy such that the equals contract is respected.

Equality among objects from different types must be compared on the basis of the same set of properties. The implementation in Figure 1.9 satisfies this requirement, but that of Figure 1.1 does not.

Java's design for equality can be restrictive. When an object needs to participate in more than one hash table as a key under distinct notions of equality, distinct equality tests are needed to work with each hash table. But the object can define only one equality test (either via the *equals()* method in its class or by inheriting one). One way to work around this limitation is to design a wrapper class with a pair of *equals()* and *hashCode()* methods for each notion of equality and use as keys objects of the wrapper class rather than the original objects. Another way in which Java's design may become restrictive is when an equality different from the one designated for hash table is needed for comparing two objects directly. For example, a full equality may be needed to compare two objects of the same type, but a weaker, type-compatible notion of equality is provided for the objects to participate in the hash table, which is not suitable for the purpose of comparing objects. When this kind of clashes happen, a method with a different name that reflects the right intent, like *similar()* or *identical()*, must be added to the object's class instead of attempting the impossible task of overloading *equals()* for multiple purposes.

Chapter 2

Preliminary Case Study

2.1 Introduction

To understand how *equals()* is implemented and what kinds of problems actually occur in real-life code, we performed an empirical study of *equals* implementation in 4 Java projects of various sizes and domains[1]. The study was conducted semi-automatically with a combination of both tool support and manual inspection. Several static checkers were developed using Eclipse's Java code analysis API [16] to search for code patterns that potentially violate the equals contract. The checkers were helpful in quickly processing large amounts of code and directing our attention to cases that are more likely to have problems. However, the design of the checkers is not the focus of this chapter. In fact, these checkers were replaced with improved versions discussed in Chapter 3.

	JDK1.5	**Lucene 2.4**	**BCEL**	**SCL**
KLOC	2552	88	39	22.6
#packages	571	14	8	29
#classes	12400	752	333	386
#interfaces	1743	26	35	11
#equals	624	40	20	15

Table 2.1: Summary of case study projects.

[1]This chapter in-part has been published in [37].

Problem	Summary	Section
Inheritance for implementation reuse	5 hierarchies	2.2.1
Equals for other purposes	9 classes	2.2.2
Type-incompatible equality	10 hierarchies	2.2.3
Type-compatible equality	Several implementation of Map.Entry	2.2.4
Hybrid equality	Map and List hierarchies	2.2.5
Evolution	Rectangle2D and Point2D	2.2.6
super.equals()	98	2.2.7
Type casting	30	2.2.8
Null checking	26	2.2.8

Table 2.2: Summary of inspected equals()-related problems in JDK 1.5.

Table 2.1 provides some overall measures of the 4 projects, JDK 1.5, Apache Lucene 2.4 [4], BCEL [6], and SCL [24]. Apache Lucene is a full text search engine, BCEL is a Byte Code Engineering Library, and SCL is a static analysis tool. JDK 1.5 is the largest project (2552 KLOC, 12400 classes) and SCL (22.6 KLOC, 386 classes) is the smallest in the group. These projects make use of the collection framework and rely on the correct implementation of *equals()* to function properly.

2.2 Detected Problems

We find that JDK 1.5 is the most representative among the 4 projects in terms of the diverse *equals()* related issues that are exhibited. We have seen a total of 174 suspicious implementations of *equals()*, which cover a wide variety of areas such as collections, utility classes, security, object broker protocol, component model, network management, compiler, GUI and image processing, and naming services. In this section, we discuss the nature, possible cause, and possible solution to the problems using JDK1.5 as a primary source of examples. We provide both class names and package names so that interested readers can verify with the JDK source code themselves. Table 2.2 provides a summary of the problems in JDK 1.5 as well as the sections where further details can be found.

2.2.1 Using inheritance for implementation reuse

When a class hierarchy is used for implementation reuse instead of subtyping [31], some problems with *equals()* may be detected. The benefit of implementation reuse is, of course, that one does not have to rewrite similar code again. But using inheritance for implementation reuse overloads the same mechanism with two purposes. When there is not documentation of intent, it can be hard to tell which one, subtyping or reuse, is intended just from the code. It becomes even worse when one part in a hierarchy is used for subtyping and another for implementation reuse. *DefaultCaret* and its superclass *Rectangle* provide such an example.

The *Rectangle* class (*java.awt*) and its superclasses, *RectangularShape* and *Rectangle2D* (*java.awt.geom*), form a type hierarchy, and implement a type-compatible equality. The relation among them is subtyping, and a *Rectangle* can behave like a *Rectangle2D* and a *RectangularShape*. However, when the *DefaultCaret* class in the *javax.swing.text* package is added as a subclass of *Rectangle*, the symmetry is lost in the equality implementation because *DefaultCaret* implements the reference equality as follows:

```
/**
 * Compares this object to the specified object.
 * The superclass behavior of comparing
 * rectangles is not desired, so this is changed
 * to the Object behavior.
 * ... (Remainder documentation omitted)
 */
public boolean equals(Object obj) {
      return (this == obj);
}
```

As can be seen in the Javadoc above, clearly, the designer of *DefaultCaret* is overriding *equals()* with a clear intention. However, this *equals()* violates the symmetry property as follows:

```
Rectangle r = new Rectangle();
DefaultCaret c = new DefaultCaret();
r.equals(c); // returns true
c.equals(r); // returns false
```

The root cause of this problem is using inheritance as implementation reuse. *DefaultCaret*, as its name suggests, implements a caret in a text box. When a caret is moved to a new location, the area (a.k.a. bound box) where it was displayed last time needs to be tracked and repainted. A bounding box happens to be a rectangle and *DefaultCaret* was made a subclass of *Rectangle* so that it can inherit and use four instance variables *x*, *y*, *width*, and *height*. In addition, it appears that these variables are intended to be private to *DefaultCaret*. However, *Rectangle* contains methods that can change them (e.g., *setRect(x,y,w,h)*). Therefore, in this case, the benefit of reuse seems to be minor in comparison with the cost of documenting and ensuring that these inherited methods must not be applied to *DefaultCaret*. A better design would be for *DefaultCaret* to compose rather than inherit *Rectangle*.

There are several well-known examples of (improper) implementation reuse in JDK, including *Vector*/*Stack* and *Date*/*Timestamp*. Our checker did not report *equals*-related problem for *Vector*/*Stack* because they belong to the *AbstractList* hierarchy, which implements a type-compatible equality. The onus is on the developer to ensure that a vector and a stack are never compared for equality, and mutators unique to a vector must not be applied to a stack. Our checker does detect a symmetry problem between *Date* and *Timestamp* (which extends *Date* and adds a nanosecond field). The author of *Timestamp* documents the intention of reuse implementation and cautions that *Timestamp* should not be substituted for *Date*.

In addition to these well-known examples, our checker also detects some new instances of implementation reuse from JDK 1.5. A class *MirrorImpl* is made the root of a large inheritance hierarchy in Java Debugging API (*com.sun.tools.jdi*) just so that all the subclasses can have a field to represent the Java virtual machine they are interacting with. A subclass *BuddhistCalendar* is derived from *GregorianCalendar*, but it appears not suitable to substitute *BuddhistCalendar* for *GregorianCalendar*. However, this is not documented. In contrast, *BakedArrayList* (*sun.swing*) is intended for reusing the implementation of *ArrayList* and the author clearly documented this intention in the code.

Although it may be possible to infer only from code whether inheritance is used for implementation reuse or subtyping, the inference is not always straightforward. For example, by inspecting the code, we conclude that *DefaultCaret* is not used as a *Rectangle*. But the process of drawing this conclusion is costly, and every future maintainer would need to repeat the same reasoning to ensure that *DefaultCaret* is not used as a *Rectangle*. The reasoning is not local and needs to be enforced whenever code changes. Thus, making *DefaultCaret* a subclass of *Rectangle* is not a good idea. If implementation reuse is at all justified, the developer should at least consider explicitly documenting the intent. In general, it appears that this documentation practice is performed better in the public API of JDK than its private part.

```
public boolean equals(Object o) {
    if (o == this) { return true; }
    else if (o instanceof IdentityHashMap) {
        IdentityHashMap m=(IdentityHashMap) o;
        // for each pair of (key, value) in m
        // test this.containsMapping(key,value)
    } else if (o instanceof Map) {
        // use value-based comparison
    } else { return false; }
}

// returns true if a pair p exists in this map
// such that p.key==key && p.value==value
private boolean containsMapping(Object key,
        Object value) {
        ...
    }
}
```

Figure 2.1: equals() of IdentityHashMap, which compares key-value pairs with == rather than equals() (with modificatons).

Map is an important type hierarchy in the Java Collection Framework, with concrete implementations such as *HashMap* and *TreeMap* that differ in implementation strategy, e.g., a hash table based map versus a tree-based map. In the *Map* interface, it is specified that the *equals()* method returns true if the given object is also a map and the two maps represent the same mappings. More formally, two maps *t1* and *t2* represent the same mappings if *t1.entrySet().equals(t2.entrySet())*. This ensures that the *equals()* method works properly across different implementations of the Map interface. As a result, a type-compatible equality is implemented in the abstract class *AbstractMap*, which is inherited by other maps.

A new class *IdentityHashMap* added in Java 1.4, however, violates the symmetry prop-

```
protected boolean equalsDelegate(Object o) {
    if (o.getClass().equals(getClass()){
        IdentityHashMap m=(IdentityHashMap) o;
        // for each pair of (key, value) in m
        // test this.containsMapping(key,value)
    }
    return false;
}

// returns true if a pair p exists in this map
// such that p.key==key && p.value==value
private boolean containsMapping(Object key,
     Object value) {
      ...
    }
}
```

Figure 2.2: equalsDelegate() for IdentityHashMap as part of implementing a hybrid equality for the Map hierarchy.

erty of the equals contract, as shown in the following snippet:

```
Map hMap = new HashMap();
Map ihMap = new IdentityHashMap();
ihMap.put("1", new Integer(1));
hMap.put("1", new Integer(1));
hMap.equals(ihMap); // returns true
ihMap.equals(hMap); // returns false
```

This loss of symmetry is caused by a change in the behavior of the *equals()* in *IdentityHashMap*, which compares keys and values with reference equality (==) rather than value equality. Thus, *hMap.equals(ihMap)* returns *true* because *hMap* is a *HashMap* and uses value equality, but *ihMap.equals(hMap);* returns *false* because *IdentifyHashMap* uses reference equality to compare pairs between itself and another map. Figure 2.1 depicts the details of *equals()* in *IdentityHashMap*.

At the time of developing *IdentityHashMap*, clearly this violation was noticed and was treated as an exception, which is evident by the following comment highlighted in bolded text in the *IdentityHashMap* specification:
This class is not a general-purpose Map *implementation! While this class implements the* Map *interface, it intentionally violates Map's general contract, which mandates the use of the* equals() *method when comparing objects. This class is designed for use only in the rare cases wherein reference-equality semantics are required.*

There can be two possible fixes for this problem, both of which are easy to implement.

The first solution is to change the *Map* hierarchy from type-compatible equality to hybrid equality in the way illustrated in Figure 1.8. This would involve modifying the *equals()* in *AbstractMap* (not shown) and adding *equalsDelegate* to *IdentifyHashMap*, which is shown in Figure 2.2. The second solution would be to move *IdentityHashMap* out of *Map* to create an independent hierarchy. This can be done by copy-and-pasting *Map* to create a new interface *IdentityMap*, and *AbstractMap* to create a new class *AbstractIdenityMap*. *IdentityHashMap* should be changed to inherit *AbstractIdentityMap* rather than *AbstractMap*. The *entrySet()* of *IdentityHashMap* and the *equals()* of *AbstractIdentityMap* also need to be modified. We have implemented a new *IdentityHashMap* in this way in a few hours. However, a potential problem with the second solution is that it separates *IdentityHashMap* from the *Map* abstraction, and thus makes it impossible for an *IdentityHashMap* to participate in client code written in terms of *Map*. If this is proven undesirable, the first solution could still be used instead.

2.2.2 Overloading equals with multiple purposes

The *equals()* contract as defined in *java.lang.Object* can support only one specific notion of equality. Sometimes, a class may need to support additional equality or some kind of *similarity* that is not intended to be used by a client such as a collection data type and even does not have to be an equivalence relation. Intentionally or incidentally, developers tend to overload the *equals()* implementation to encode all of them in one place. For ease of understanding and maintenance, it is advisable to implement separate predicates for such relations.

For example, the *String* and *StringBuffer* classes (*java.lang*) implement the *CharSequence* interface and represent a sequence of characters. The difference is that *String* is immutable and *StringBuffer* is mutable. *equals()* in *String* will always return false when compared with a *StringBuffer*. Since objects from these classes contain character sequences, it makes sense to test if the content of a *String* object is the same as that of a *StringBuffer*. Instead of piggybacking onto *equals()*, the *String* class provides a "*con-*

tentEquals(StringBuffer)" method for this purpose, which is not symmetric with respect to *StringBuffer*.

The *Arg* class represents an argument on the stack (*com.sun.org.apache.xpath.internal*). Among other fields, an *Arg* has a qualified name *QName* (*com.sun.org.apache.xml.internal.utils*).

```
public class Arg {
 private QName m_qname;
 public boolean equals(Object obj) {
  if (obj instanceof Qname)
   {return m_qname.equals(obj);}
  else return super.equals(obj);
 }
 ... // Remainder omitted
}
```

equals() in *Arg* directly checks for the type of *obj* (*obj instanceof QName*) and compares it with *m_qname*. Note that *QName* does not belong to the type hierarchy of *Arg*. Furthermore, if *obj* is not of type *QName* (including objects of *Arg* class), *equals()* will perform a reference equality check. Though the *equals()* method of *Arg* checks for *QName*, *equals()* in *QName* does not check for *Arg*, which means a loss of symmetry between the two classes. Thus, the *equals()* definition is probably intended to serve as a similarity check, and a better solution would be to remove the *equals()* method from the *Arg* class (in which case it will inherit Object's *equals()*) and add a new method as follows:

```
public boolean hasName(QName qName) {
  return m_qname.equals(qName);
}
```

There are several ways to detect such cases of piggybacking on *equals()*. A particularly useful means is to detect the presence of multiple *instanceof* type testing. Our experience is that most *equals()* implementations contain only one *instanceof*. Therefore, when an *equals()* contains more than one *instanceof* testing, it is more likely that the *equals()* predicate is overloaded to carry another notion of 'similarity'. A checker was written to detect the presence of multiple *instanceof* and 15 such cases were found in JDK 1.5. We inspected all of them and concluded that 9 of them are true positives and 6 false positives.

The *AlgorithmId* class in *sun.security.x509* represents algorithms such as cryptographic transformations. Its *equals()*, as shown below, clearly overloads itself to carry both an equality check and a similarity check with *ObjectIdentifier* in *sun.security.util*.

A similar case occurs between the *Oid* class in *org.ietf.jgss* and *ObjectIdentifier*.

```
public boolean equals(Object other) {
if (this == other) {
    return true; }
if (other instanceof AlgorithmId) {
    return equals((AlgorithmId) other);
} else if (other instanceof ObjectIdentifier) {
    return equals((ObjectIdentifier) other);
} else {
    return false;}
}
```

Two classes *GroupImpl* and *NetmaskImpl* (com.sun.jmx.snmp.IPAcl) in the *java.security .Principal* hierarchy mistakenly encode a partial order similarity between 2 subnet masks in the *equals()* predicates (255.255.255.0 is 'equal' to 255.255.0.0 but not vice versa).

The *XObject* hierarchy in *com.sun.org.apache.xpath.internal.objects* shows that the developer is not clear about how to implement a sophisticated equality, under which, for example, a string (*XString*) that represents a number would be considered equal to another object that represents a true number. The strategy appears to be for *XString.equals()* to invoke *equals(Object)* from 2 other classes. Because only *XString* overrides *equals(Object)*, and all the other classes in the hierarchy implement *equals(XObject)* instead, we conclude that the overriding of *equals()* in *XString* is incidental and that it is not intended to conform to the contract of *java.lang.Object.equals()*.

2.2.3 Suspicious implementations of type-incompatible equality

The most common pattern found in JDK is probably implementing a type-incompatible equality by using an *instanceof* test in both a supertype and a subtype. In the example shown in Figure 2.3, *NTSid* represents a Security Identifier for Windows NT OS, which has 4 concrete subclasses that model a user, a domain, a group, and a primary group,

respectively. These are principals that can be attached to a subject such as a person to grant the subject a certain permission.

```
// implementation of NTSid
public boolean equals(Object o) {
   if (o == null) return false;
   if (this == o) return true;

   if (!(o instanceof NTSid))
            return false;
   NTSid that = (NTSid)o;

   if (sid.equals(that.sid)) {
        return true;
   }
   return false;
}

// implementation of NTSidUserPrincinpal
public boolean equals(Object o) {
     if (o == null)       return false;
     if (this == o)       return true;

     if (!(o instanceof NTSidUserPrincipal))
         return false;

     return super.equals(o);
}
```

Figure 2.3: A common mistake of using instanceof to implement type-incompatible equality in a supertype (NTSid) and a subtype (NTSidUserPrincinpal).

Because *NTSid* and its subclasses are logically distinct objects, a type-incompatible equality should be implemented for this type hierarchy. Unfortunately, the solution presented in Figure 2.3 is suspicious as it breaks the symmetry property between *NTSid* and its subclasses. Fortunately, this can be fixed with the type-incompatible implementation shown in Figure 1.5. Another possibility is that *NTSid* is not intended to be instantiated. If that is the case, then at least it could have been made an abstract class. Furthermore, its *equals()* can be removed to make it clear that a type-incompatible equality is intended for this hierarchy.

Table 2.3 shows other similar cases we found from JDK 1.5. Note that these are all type hierarchies and a much larger number of classes are involved.

2.2.4 Suspicious implementations of type-compatible equality

When a type hierarchy is implemented by multiple developers, it can be easy for some to forget about the programming disciplines.

package	root class
com.sun.java_cup.internal	lr_item_core
com.sun.jndi.ldap	ClientId
com.sun.security.auth	NTSid
java.beans	PropertyDescriptor
java.security	CodeSource
javax.management	MBeanFeatureInfo
javax.management	MBeanInfo
javax.imageio	ImageTypeSpecifier
com.sun.jmx.snmp.IPAcl	PermissionImpl
java.awt.image	ColorModel

Table 2.3: Class hierarchies in JDK 1.5 that implement type-incompatible equality with instanceof.

The Map interface defines a map entry (key-value pair) to model mapping from a key to a value. Formally, two entries e1 and e2 represent the same mapping if the following holds:

```
(e1.getKey()==null ?
 e2.getKey()==null:
   e1.getKey().equals(e2.getKey()))  &&
(e1.getValue()==null ?
 e2.getValue()==null:
   e1.getValue().equals(e2.getValue()))
```

This ensures that equals() works properly across implementations of the *Map.Entry* interface.

Figure 2.4 shows both a good implementation of *equals()* in *Hashtable*'s map entry, which accepts *Map.Entry*, and an inappropriate implementation in *java.text.AttributeString*, which accepts only *AttributeEntry*. Another inappropriate implementation of map entry can be found in *ParserImplTableBase* (*com.sun.corba.se.spi.orb*). Clearly, these developers either do not know or forget to follow the right pattern in Figure 2.4. Although it is not certain whether the inappropriate type casting would lead to a problem, it would be a good idea to conform to the standard implementation shown in *Hashtable*.

```
// implementation in java.util.Hashtable
public boolean equals(Object o) {
    if (!(o instanceof Map.Entry))
       return false;
    Map.Entry e = (Map.Entry)o;

    return (key==null ? e.getKey()==null :
    key.equals(e.getKey())) &&
    (value==null ? e.getValue()==null :
    value.equals(e.getValue()));
}

// implementation in java.text.AttributeEntry
public boolean equals(Object o) {
     if (!(o instanceof AttributeEntry)) {
        return false;
     }
    AttributeEntry other = (AttributeEntry) o;
    return other.key.equals(key) &&
    (value == null ? other.value == null :
    other.value.equals(value));
}
```

Figure 2.4: A mistake of type testing the wrong type when implementing type-compatible equality.

2.2.5 Hybrid equality

Initially, the *Map* type hierarchy is designed to have a type-compatible equality. An abstract class *java.util.AbstractMap* provides the skeleton implementation for most of the operations specified in the *Map* interface, including the *equals()* method. *AbstractMap* is then extended by other maps.

However, not all subclasses are type-compatible with *AbstractMap*. For example, the Java Debugging API (*com.sun.tools.jdi*) defines a *LinkedHashMap* whose *equals()* requires its parameter to be the same type by an *instanceof* testing. It is also noticed that this class is very similar to a same named class in *java.util*, which inherits *equals* from *AbstractMap*. Thus it seems reasonable to conclude that the jdi *LinkedHashMap* is intended to be type-incompatible with *AbstractMap*. Two other similar cases are *RenderingHints* (*java.awt*) and *TabularDataSupport* (*javax.management.openmbean*), which implement *Map* and should be type-incompatible with other maps due to domain semantics. Again, these two classes also use *instanceof* to implement type-incompatible equality, which can be better done by following the implementation shown in Figure 1.8.

The *List* hierarchy has similar subclasses that are intended to be type-incompatible. For example, *com.sun.corba.se.impl.ior* defines a list called *FreezableList* that is intended to be

type-incompatible with other lists. Another class *BakedArrayList* from *sun.swing* is also type-incompatible, which specializes *ArrayList* for better performance. Although its author documents that *BakedArrayList* is for local use only and thus its *equals()* implementation is good enough, changing the *List* hierarchy into hybrid equality would make this class exhibit the general *equals()* behavior, which will ease future maintenance since the general behavior can always be assumed and one does not have to worry whether objects of this class are compared with other general lists.

2.2.6 Evolution of type hierarchy

When a type hierarchy is evolved, it needs to be revisited to ensure that the right equality is implemented properly.

The state of *Rectangle2D* (*java.awt.geom*) can be specified by 4 methods *(getX(), getY(), getWidth(), getHeight())*, all return *double*. The *Rectangle* class (*java.awt*) is a subclass of *Rectangle2D* that uses *int* as its representation. (There are several other subclasses using other representations like *float* and *double*.) In theory, a type-compatible equality can be implemented for this type hierarchy by defining an *equals()* in *Rectangle2D*. However, as shown in Figure 2.5, *Rectangle* also defines an *equals()* in addition to that of its superclass *Rectangle2D*.

The implementation of *equals()* in *Rectangle* is redundant and can be removed. By inspecting the code for *Rectangle* in JDK 1.0, we found that *Rectangle* exists before *Rectangle2D* (added in JDK 1.2). After the *Rectangle2D* class was introduced in JDK 1.2, *Rectangle* was retrofitted to extend *Rectangle2D*. As a result, methods like *getX()* was added to *Rectangle*. Furthermore, as a quick fix, most of the original *equals()* was kept and a *super.equals()* call was added.

A similar case happens between *java.awt.geom.Point2D* and *java.awt.Point*.

A notable pattern in the evolution of type hierarchy is to specialize a general class for local use. The local use may relax from the subclass some of the restrictions put on the general class. For example, it may be guaranteed that 'the subclass may never interact

```
public abstract class Rectangle2D
 extends RectangularShape {
  /** ... @since 1.2 */
  public boolean equals(Object obj) {
   if(obj == this) { return true; }
   if (obj instanceof Rectangle2D) {
    Rectangle2D r2d=(Rectangle2D)obj;
    return((getX() == r2d.getX()) &&
     (getY() == r2d.getY()) &&
     (getWidth() == r2d.getWidth()) &&
     (getHeight() == r2d.getHeight()));
   }
   return false;
  }
  ... // Remainder omitted
}

/** ... @since JDK1.0 */
public class Rectangle extends Rectangle2D
 implements Shape, Serializable {
  public int x;
  public double getX() { return x; }
  ... // Remainder getters omitted
  public boolean equals(Object obj) {
   if (obj instanceof Rectangle) {
    Rectangle r = (Rectangle)obj;
    return ((x == r.x) && (y == r.y) &&
     (width == r.width) && (height == r.height));
   }
   return super.equals(obj);
  }
  ... // Remainder omitted
}
```

Figure 2.5: equals() in Rectangle2D and Rectangle.

with other general classes in the same hierarchy'. However, because the truth of such properties depends on the context of use, it can be costly to enforce. If such local properties are desired, they should at least be documented, as they were in *java.sql.Timestamp* and *sun.swing.BakedArrayList*.

2.2.7 Implementation variations

In this section, common ways of implementing *equals()* are summarized and evaluated, particularly, calling super.equals() and type testing operations, and advice is given for their use.

We suggest to avoid calling *super.equals()* in the *equals()* method unless absolutely necessary. For each class in a hierarchy, define its state first, including those inherited from its superclasses, and implement *equals()* in terms of the state independent of any superclass. In this way, it becomes easier to understand *equals* as everything it depends on is presented

in a single location. Furthermore, a subclass gains the maximum independence from the superclass because *equals* depends on state instead of representation. Of course, this would imply that more code needs to be typed, especially for deep class hierarchies. But our experience is that most *equals()* are short.

To understand how *super.equals()* is called in practice, a checker was developed and 98 classes in JDK 1.5 are detected whose *equals()* call *super.equals()*. 72 out of 98 are for implementation reuse, most of which could be changed to follow the above advice easily. 11 out of 98 end up calling *Object.equals()*, and thus are redundant. In the example that follows, the super call should be replaced with *false*.

```
class Point2D { // java.awt.geom
...
public boolean equals(Object obj) {
  if (obj instanceof Point2D) {
      Point2D p2d = (Point2D) obj;
      return (getX() == p2d.getX()) &&
      (getY() == p2d.getY());
  }
  return super.equals(obj);
}
```

However, there is one special case where a superclass makes all its instance variables private but does not define accessors. In such cases, *super.equals()* would be justified. Finally, 5 out of 98 super calls are used in a subclass whose *equals()* is semantically equivalent to that of the superclass but differs in performance or the representation used. For example, *EnumMap* is a map whose keys are enum. It provides a specialized implementation for *equals()* using its own representation. When the incoming argument is not *EnumMap*, the superclass' *equals()* is called. This represents another condition where *super.equals()* is justified in a subclass.

Table 2.4 depicts the use of various type testing operations in the 4 projects. *instanceof* as a type checking mechanism is more popular than others such as *getClass()*, *Type.class* and *try-catch*. The use of the *instanceof* operator seems to be the norm for *equals()* implemen-

	JDK	Lucene	BCEL	SCL
instanceof	496	32	13	9
getClass	26	5	0	5
Type.class	0	2	0	0
try-catch	37	1	0	0
none	65	0	7	1
multiple testing	15	0	0	0

Table 2.4: Use of type testing in equals().

tation for most projects. This would explain why there are so many violations of symmetry when *instanceof* is used to implement type-incompatible equality between a supertype and a subtype (Section 2.2.3), where *getClass()* should have been used instead.

2.2.8 Other design considerations

Two implementation details of *equals()* need to be considered. One is that before dereferencing the incoming parameter, it should be checked not to be null. The other is before casting the parameter, it must be tested that it can indeed be cast to the given target type. Using two checkers, we were able to conclude for JDK 1.5 26 cases of possible null pointer dereferencing and 30 cases of inappropriate type casting that may result in an exception.

Consider *equals()* of *SegmentInfo* (*org.apache.lucene.index*).

```
public boolean equals(Object obj) {
 SegmentInfo other;
 try{ other = (SegmentInfo) obj;}
 catch(ClassCastException cce){return false;}
 return other.dir==dir&&other.name.equals(name);
}
```

This *equals()* checks for *ClassCastException* but fails to check for a null value of parameter *obj*. Thus it may throw a *NullPointerException*. The author of this class may be relying on a local assumption that *obj* is never null, but since the cost of proper implementation is so little, it would be worthwhile to fix it so that one does not have to rely on this assumption.

The *org.apache.bcel.classfile.Field* class from the BCEL project is shown in Figure 2.6. Method *equals(Object, Object)* in the comparator does not check for the type of *o1* and *o2* before casting, and thus may throw *ClassCastException*. Furthermore, it also dereference

```
public final class Field extends FieldOrMethod{
private static BCELComparator _cmp =
  new BCELComparator() {
public boolean equals(Object o1,Object o2){
 Field THIS = (Field) o1;
 Field THAT = (Field) o2;
 return THIS.getName().equals(THAT.getName())
 &&THIS.getSignat().equals(THAT.getSignat());
}
   ... // Remainder of BCELComparator omitted
};
public boolean equals( Object obj ) {
  return _cmp.equals(this, obj);
 }
  ... // Remainder of Field class omitted
}
```

Figure 2.6: equals() of Field class.

THIS and *THAT* without checking for *null*, and thus may result in *NullPointerException*. Again, the author may rely on a local assumption that *o1* and *o2* are of the right type, but since the cost of proper implementation is so little, it would be worthwhile to fix it so that one does not have to rely on this assumption.

As a final example, consider *equals()* shown below (*com.sun.org.apache.xalan.internal .xsltc.compiler.FunctionCall.JavaType*):

```
static class JavaType {
     public Class  type;
     public int distance;

     public JavaType(Class type, int distance){
            this.type = type;
            this.distance = distance;
     }
     public boolean equals(Object query){
          return query.equals(type);
     }
}
```

Objects of this class are used as a value in a *Map* and are compared with objects of *Class* directly using *aJavaType.equals(aClassObject)* This solution works correctly with the current *Map* implementation. But again, this use is local and it relies on the assumption that *Map* compares two objects with *JavaType* as a receiver object, which would be an extra burden for a future maintainer to ensure.

2.3 Equals Implementation Guidelines

Based on the analysis in Section 1.3 and the case study in Section 2.1, we recommend the following guidelines for designing, implementing, and evolving *equals()*. The correct implementation of *equals()* requires proper identification of object state and designing the right type hierarchies. In this respect, our design guidelines for *equals()* are consistent with established principles for designing type hierarchy [30, 31]. But our guidelines are also equals-specific and require the developer to apply the right implementation strategy.

1. Identify state for each class in a hierarchy.

2. Use inheritance for subtyping rather than implementation reuse. If implementation reuse is used for a good reason and cannot be avoided, document it.

3. Decide the right equality that is needed for an inheritance hierarchy (type-compatible, type-incompatible, or hybrid equality) and use the corresponding implementation pattern.

4. Avoid piggybacking on *equals()* to implement relations other than equality. Add other predicates instead.

5. Minimize the dependency on superclass. Consider implementing *equals()* in terms of state (e.g., accessors) rather than internal representation. Avoid calling *super.equals()* whenever possible.

6. Minimize implementation variations. For example, avoid using exception handling to test object types.

7. Avoid the possibility of *NullPointerException* and *ClassCastException* by following the proper implementation. Avoid relying on local assumptions since it is easy to provide a reliable implementation.

8. When an inheritance hierarchy is changed, reconsider all of the above.

9. Keep in mind that Java's design can support only one equality. When more than one equality is needed, consider other design options in addition to *equals()*.

Chapter 3

Static Checker for Analysis of Object Equality

3.1 Introduction

The preliminary case study from Chapter 2 convinced us that the *equals* method is not as simple as it looks and it is very easy to get it wrong. We felt a need for a static analysis tool that can properly identify the problems reported in Chapter 2. We looked at the FindBug [43] tool and found that the tool focuses more on a **call** to the *equals* method rather than its **implementation**. Out of *36 equals* related problems reported by the tool, we found only *4* categories of problems that deal with equivalence property violation in the implementation of the *equals* method. Among these only one category deals with *equals* implementation with respect to a type hierarchy. However, none of these categories properly address the violation of equivalence relation in the implementation of the *equals* method. Other available static checkers do very little with the *equals* implementation[1]. A presence of many equivalence related problems and modern day checkers not properly addressing these issues motivated us to design our own checker. Furthermore, we want the checker to be able to detect *reflexive*, *symmetric* and *transitive* property violation based on the semantics of

[1]This is further discussed in Section 5.4.

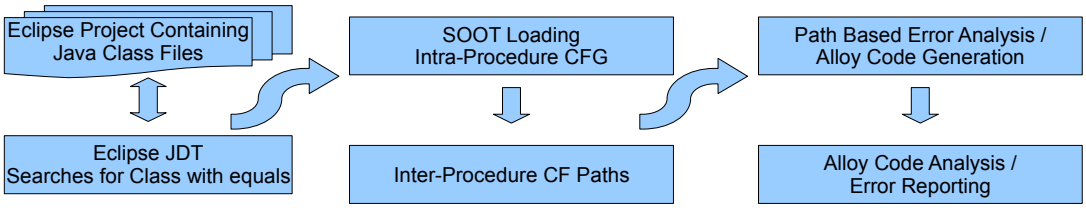


Figure 3.1: The context diagram of equals checker.

the code through logical abstraction rather than just through the analysis of the program syntax and structure.

3.2 Checker Architecture

The equals checker is a path-based analysis tool designed as an Eclipse [17] plugin. It works on a set of control flow paths generated for an *equals* method. This process is accompanied with a different kind of path filtering analysis for efficiency as well as for low level error detection. It then tries to find different kinds of code patterns in these paths that are then translated as constructs of the first order logic language (Alloy [26, 27]). This process abstracts out implementation details of Java code and provides a general mechanism for imposing high level logical constraints on the implementation of the *equals* method. The high level constraints for the *equals* method in this case are the *reflexive*, *symmetric* and *transitive* property of the equivalence relation discussed in Section 1.1.

The checker requires three different frameworks to achieve translation of Java to the Alloy code and execution of Alloy model for constrain checking. These frameworks are as follows:

- Eclipse's Java Development Tool
- Soot
- Alloy

Figure 3.1 shows the context diagram of the checker and its components. Eclipse's Java

Development Tools (JDT) [16] is used for searching all of the classes that override the *equals* method of *java.lang.Object* in a project. Type hierarchies are then computed with the help of JDT that involve these classes. All of the classes in these type hierarchies are then loaded to Soot [44] that translates the byte code of these classes into a three address stack-less code called *Jimple*. A control flow graph (CFG) built on top of Jimple is then used for Path generation. The CFG returned by Soot for an *equals* method is intra-procedural, i.e. it does not expand any method call within the body of the *equals* method. Hence, we had to develop our own library on top of Soot for Inter-procedural path generation and path based context-sensitive flow-sensitive analysis. We used a modified version version of Class Hierarchy Analysis (CHA) [14] for type resolution of the target object at the call site to process expansion of intermediate method calls and extract relevant control flow paths. All of the redundant paths are pruned while processing path generation. The generated paths are then passed to a code generator for Jimple-based low level error checking as well as for Alloy code generation. The generated Alloy code serves as an abstract model of Java code that can be analyzed by Alloy analyzer for checking higher level logical constraints like *reflexive*, *symmetric* and *transitive* properties of the *equals* method. Any error discovered is then reported to the user using the Eclipse GUI.

3.3 SERL Control Flow Framework

SERL (Software Engineering Research Laboratory) Control Flow Framework built on top of Soot provides three major functionalities:

- Path generation
- Path, context and flow sensitivities in an analysis
- Code pattern detection

3.4 Path Generation

Path generation for a method is done using its flow graph called *ExceptionalUnitGraph* returned from Soot. The algorithm at first generates a set of intra-procedure paths that do not expand method calls. It then iterates through all of the statements in the generated set of paths looking for a method call. Each method call is then expanded to get a new set of intra-procedure paths. All of the newly generated paths are appropriately in lined with previously existing paths. After all of the method calls in the path set are expanded, the final set of paths are now inter-procedure paths whose method calls are expanded wherever needed.

3.4.1 Intra-procedure Path Generation

The algorithm (Figure 3.2) for intra-procedure path generation uses breadth first search for traversing nodes in a control flow graph. The algorithm takes a directed flow graph $G = (V, E)$ as input and outputs a set of paths S containing a list of vertices. The algorithm assumes that the graph has one entry vertex, may have multiple exit vertices and may contain cycles.

Let us apply the algorithm to a Java code snippet with loops (cycles) shown in Figure 3.3 whose flow graph is shown in Figure 3.4. Before the first loop starts Q has the {0} node in it and S also has one path with the 0 node in it {[0]}. Inside the first loop n = 0, Q = {}, W = [0], O = [0] and s = [1] (since successors(0) = [1]). The condition i == 0 is satisfied and the algorithm iterates through all of the available paths, in this case just one p = [0]. It checks for duplicate node 1 in [0] resulting in c = 0. So, condition c <= 1 is satisfied and 1 is appended to path [0], making it [0,1]. 1 is also added to Q which now becomes {1}.

In the second iteration, n = 1, Q = {}, W = {[0,1]}, O = {[0,1]} and s = [7,2]. The condition i = 0 is satisfied for s[0] = 7, and the algorithm proceeds like it did in the first iteration resulting in S = {[0,1,7]}. However, for s[1] = 2, i == 0 is not satisfied and the else block is executed. There is no duplicate node 2 in O = {[0,1]}, so, c <= 1 is satisfied. Node 2 is appended to a mutable clone of p = [0,1] and the clone is added to S. At the

```
S IntraProcedurePathGenerator(G) {
  Q = {} // An empty queue that does not allow duplicate entries
  S = {} // An empty path set
  t = [] // An empty list of graph nodes

  r = root node of G
  add r to t  // Add root node to empty path
  add t to S // Add first path to the path set

  add r to Q // Add root node to the queue
  while(Q is not empty) {
    n = poll Q
    W = set of paths ending with n; // Mutable working copy of intermediate paths
    O = clone W; // Immutable copy of W
    s = successors(n) // List of successors of n
    for(i = 0 to length(s) - 1) {
      if(i == 0) {
        // For first child use mutable working copy of intermediate paths
        // This block does not change the size of S
        for(each path p in W) {
          c = number of duplicate s[i] in p
          if(c <= 1) { // Prevents entering a loop more than two times
            append s[i] to p
            add s[i] to Q
          }
        }
      }
      else {
        // For rest of the children, use immutable copy and clone to get a mutable copy
        // This block may add a new path to S thus increasing its size by 1
        for(each path p in O) {
          c = number of duplicate s[i] in p
          if(c <= 1) { // Prevents entering a loop more than two times
            d = clone p // Get mutable clone of p
            append s[i] to d
            add d to S
            add s[i] to Q
          }
        }
      }
    }
  }

  // Remove paths from S whose last node is not a return/exit node
  for(each path p in S) {
    l = last node of p
    if(l has non empty successors) {
      remove p from S
    }
  }
  return S
}
```

Figure 3.2: Intra-procedure path generation algorithm.

```
public static int sampleMethod() {
0 int i = 0;
1 while(i < 5) {
2  int j = 2 * i;
3  while(j < 20) {
4    if(j == 4) break;
5    j = j + i;
   }
6  i = i + 1;
  }
7 return i;
}
```

Figure 3.3: A Java code snippet showing loops and inner loops.

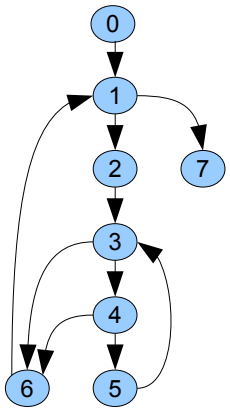


Figure 3.4: Flow graph corresponding to code snippet shown in Figure 3.3.

end of the second iteration S = {[0,1,7], [0,1,2]} and Q = {7,2}. When condition i == 0 is satisfied, the size of S is not changed, only the corresponding path is updated, however, when i == 0 is not satisfied a new path is added to S, thus increasing the size of S by 1. In the third iteration, n = 7 which is the return node and does not have a successor, so, nothing is changed except Q = {2}. In the fourth iteration, we will get the following Q and S:

```
Q: {3}
-------------------------------------------------------
1: 0  1  7
-------------------------------------------------------
2: 0  1  2  3  (Updated)
```

In the fifth iteration, we will add another path as follows:

```
Q: {6  4}
```

```
-------------------------------------------------------
1: 0  1  7
-------------------------------------------------------
2: 0  1  2  3  6  (Updated)
-------------------------------------------------------
3: 0  1  2  3  4  (Added)
```

The algorithm proceeds in a similar way until the 13^{th} iteration where we have the following Q and S:

```
Q: {6  4  7  2}
-------------------------------------------------------
1: 0  1  7
-------------------------------------------------------
2: 0  1  2  3  6  1  7
-------------------------------------------------------
3: 0  1  2  3  4  5  3  6  *
-------------------------------------------------------
4: 0  1  2  3  4  6  1  7
-------------------------------------------------------
5: 0  1  2  3  6  1  2  3  6  *
-------------------------------------------------------
6: 0  1  2  3  4  5  3  4
-------------------------------------------------------
7: 0  1  2  3  6  1  2  3  4
-------------------------------------------------------
8: 0  1  2  3  4  6  1  2
```

In the 14^{th} iteration, n = 6 and s = [1] and W = {[0,1,2,3,4,5,3,6], [0,1,2,3,6,1,2,3,6]} (containing two paths ending with 6, i.e. paths 3 and 5 in S of the 13^{th} iteration). While calculating duplicate successor 1 in path [0,1,2,3,4,5,3,6], count = 1 (<= 1) and the path is updated but for path [0,1,2,3,6,1,2,3,6] count = 2 (> 1), hence this path is ignored. This duplicate check prevents control from processing the same node more than two times in a path. And not adding these nodes in the queue makes the algorithm stop in a finite time. The following modification happens to Q and S after the 14^{th} iteration:

```
Q: 4  7  2  1
-------------------------------------------------------
3: 0  1  2  3  4  5  3  6  1  (Updated)
-------------------------------------------------------
5: 0  1  2  3  6  1  2  3  6  (Ignored)
-------------------------------------------------------
```

Q becomes empty in the 30^{th} iteration generating following set of paths:

```
Q: {}
-------------------------------------------------------
1: 0  1  7
-------------------------------------------------------
2: 0  1  2  3  6  1  7
-------------------------------------------------------
3: 0  1  2  3  4  5  3  6  1  7
-------------------------------------------------------
4: 0  1  2  3  4  6  1  7
-------------------------------------------------------
5: 0  1  2  3  6  1  2  3  6  (will be pruned)
-------------------------------------------------------
6: 0  1  2  3  4  5  3  4  5  (will be pruned)
-------------------------------------------------------
7: 0  1  2  3  6  1  2  3  4  5  (will be pruned)
-------------------------------------------------------
8: 0  1  2  3  4  6  1  2  3  6  (will be pruned)
-------------------------------------------------------
9: 0  1  2  3  4  5  3  4  6  1  7
-------------------------------------------------------
10: 0  1  2  3  6  1  2  3  4  6  (will be pruned)
-------------------------------------------------------
11: 0  1  2  3  4  5  3  6  1  2  (will be pruned)
-------------------------------------------------------
12: 0  1  2  3  4  6  1  2  3  4  5  (will be pruned)
-------------------------------------------------------
13: 0  1  2  3  4  5  3  4  6  1  2  (will be pruned)
-------------------------------------------------------
14: 0  1  2  3  4  6  1  2  3  4  6  (will be pruned)
```

Note that there are some intermediate paths that were not processed due to existing duplicate entries in them. The algorithm removes all of these intermediate paths that do not end at return nodes (node 7 in this case), resulting in the following final set of intra-procedure paths:

```
1: 0  1  7
-------------------------------------------------------
2: 0  1  2  3  6  1  7
-------------------------------------------------------
3: 0  1  2  3  4  5  3  6  1  7
-------------------------------------------------------
```

```
4: 0  1  2  3  4  6  1  7
-------------------------------------------------------
5: 0  1  2  3  4  5  3  4  6  1  7
```

As we can see in the final set of paths, a loop is converted into two paths, one that does not go inside the loop when the loop condition is unsatisfied (e.g. path 1) and another when the loop condition is satisfied (e.g. path 2). After generating intra-procedure paths, inter-procedure paths that expands intermediate method calls are generated. This is discussed in the following section.

3.4.2 Inter-procedure Path Generation

The Inter-procedure path generation algorithm is essentially a work list algorithm. It uses the algorithm from intra-procedure path generation (Figure 3.2) to get a first set of paths and iterates through the statements of these paths, expanding methods intra-procedurally and in-lining with the existing paths until all of the method calls are *selectively* expanded. Note that we run several analyses on the intermediate paths to determine if a method needs to be expanded (Section 3.4.3) and expand only those that need to be expanded.

All of this functionality is based upon the Jimple representation of Java code. To get a little understanding of Java to Jimple code transformation, lets look at the *equals* method of a *Point* class shown in Figure 3.5. Line 1 and 2 are field declarations. Line 3 serves as a method header for the *equals* method which takes a parameter of type $java.lang.Object$ and returns a boolean value. Line 4 invokes the *typeCheck* method defined in line 9 that performs $instanceof$ type checking and returns true if o is of type *Point* or its subtypes. Line 6 does *Object* to *Point* type casting and line 7 returns the result of comparison of x and y states of *this* and o objects.

The transformation of *Point's equals* method to Jimple is shown in Figure 3.6. Since Jimple is a stack-less representation of bytecode, all of the variables used in the equals method are declared locals including the *this* reference. Line 1 assigns the *this* reference of the *Point* class to a local variable $r0$. Line 2 assigns parameter o to $r1$. Line 3 assigns the result of $this.typeCheck(o)$ to $z0$. Line 4 compares the value of $z0$ not to be *false* (i.e.

```
public class Point {
1   private int x;
2   private int y;

3   public boolean equals(Object o) {
4     if(!this.typeCheck(o))
5       return false;
6     Point that = (Point) o;
7     return this.x == that.x && this.y == that.y;
8   }

9    public boolean typeCheck(Object o) {
10    if(o instanceof Point)
11      return true;
12    return false;
13   }
}
```

Figure 3.5: Point Class containing equals method.

true), in which case, *goto* statement will result in line 6 to be executed that does type casting of *Object o* to *Point that* and stores in *r2* local. If the result of the comparison is false then *return false* of line 5 will be executed. Line 7 and 8 assign the field x of *this* and *that* to *i0* and *i1* respectively. Line 9 compares *this.x* and *that.x*, if they are not equal, line 14 will be executed and the program will terminate with *return false*. If the comparison evaluates to true then line 10 will be executed where *i2* and *i3* are assigned *this.y* and *that.y* respectively. Line 12 does the comparison of *this.y* and *that.y*. If they are not equal then *return o* of line 14 will be executed otherwise *return true* of line 12 will be executed. The information of control flow is represented by a data structure called *ExceptionalUnitGraph* of Soot that is used for path generation. The internal workings of Soot and details of Jimple representation is out of the scope of this work but can be found in [44].

```
1   r0 := @this: serl.test.Point
2   r1 := @parameter0: java.lang.Object
3   $z0 = virtualinvoke r0.<serl.test.Point: boolean typeCheck(java.lang.Object)>(r1)
4   if $z0 != 0 goto r2 = (serl.test.Point) r1
5   return 0
6   r2 = (serl.test.Point) r1
7   $i0 = r0.<serl.test.Point: int x>
8   $i1 = r2.<serl.test.Point: int x>
9   if $i0 != $i1 goto return 0
10  $i2 = r0.<serl.test.Point: int y>
11  $i3 = r2.<serl.test.Point: int y>
12  if $i2 != $i3 goto return 0
13  return 1
14  return 0
```

Figure 3.6: Java to Jimple Transformation of equals method.

The call site at line 3 of Figure 3.6, is not expanded by Soot. The SERL framework provides mechanism for expansion of such intermediate method calls within the context of the analysis method(in our case *equals*) per path basis.

The algorithm for inter-procedure path generation is shown in Figure 3.7. The input to the algorithm is a directed flow graph $G = (V, E)$ and set of path filters F that has logic for pruning paths and returns a set of paths S where each path contains a list of PathNodes[2].

Let us go through this algorithm for the *equals* method shown in Figure 3.6. A call to $IntraProcedurePathGenerator(G)$ of Figure 3.2 results in four paths: a first path that returns false for invalid *typeCheck* (Figure 3.8), a second and third path that return false for unequal x (Figure 3.9) and y (Figure 3.10) respectively, and a fourth path that returns true for equal x and y (Figure 3.11).

SERL framework has modules for context sensitive path-based flow analysis that can be run over these paths. Three specific analyses have been developed for the *equals* checker for path filtering: *BooleanFilter*, *NullnessFilter* and *TypeAnalysisFilter*. *BooleanFilter* performs copy propagation and constant folding for boolean values followed by dead code elimination and pruning of paths returning false. *NullnessFilter* performs copy propagation and constant (null constant) folding for reference values followed by dead code elimination. *TypeAnalysisFilter* accumulates type information for local variables and filters paths containing dead code corresponding to non-posible type checking. For the *equals* method only paths returning true are considered because anything not true is false in logic and as a good approximation of the code, false paths are pruned. This approach significantly reduces the number of paths and helps to avoid path explosion in lengthy *equals* methods with several control flow branches. $BooleanPathFilter$, in this case, will prune paths in Figure 3.8, 3.9 and 3.10 resulting in only one path (Figure 3.11) for further processing.

The algorithm now proceeds to finds a call site (*typeCheck* method) at line number 3 of Figure 3.11. The *Point* class, in our example, does not have a subclass, so the *class hierarchy analysis* results in only *Point* as a target class for *this* reference. The Jimple code

[2]A PathNode decorates each vertex(a Jimple statement) with a context object.

```
S InterProcedurePathGenerator(G, F) {
  Q = [] // An empty queue of Paths
  S = {} // Set of final Paths
  P = IntraProcedurePathGenerator(G)
  nP = JimpleStatementsToPathNodes(P)
  fP = apply F to nP // Filter newly generated paths if possible

  // Create work list by adding paths to Q
  for(each p in fP) {
    add p to Q
  } // End For
  while(Q is not empty) {
    p = poll Q
    flag = 0 // a flag denoting whether a node of p is expanded or not
    i = 0 // Sentinel for PathNode of Path p
    while(i < length(p) && flag == 0 ) {
      if(p[i] contains call site and method needs to be expanded) {
        flag = 1
        // CHA is used for resolving method call to possible target methods
        M = set of target methods resolved for call site at p[i]
        for(each m in M) {
          mG = flowgraph(m) // ExceptionUnitGraph returned from Soot
          mP = IntraProcedurePathGenerator(mG)
          mNP = JimpleSatementToPathNodes(mP)
          // Inline newly produced paths to the clone of existing path
          for(each path mp in mNP) {
            cp = clone p

            // Make call site appear before and after the in lining to denote
            // method entry and method return points in a path
            cp[i + length(mp) + 1] = clone cp[i]

            // Move previously existing nodes down by length(mp) and inline new path
            for(j = 0 to length(mp)) {
              cp[i + j + length(mp) + 2] = cp[i + j + 1]
              cp[i+j + 1] = mp[j]
            } // End For
            // Add cp to queue for further processing
            fCP = apply F to cp
            if(fCP is not empty) { // Meaning filter F did not prune path cp
              add cp to Q
            } // End If
          } // End For
        } // End For
        i = i + 1
      } // End If
    } // End While
    if(flag == 0) {
      // Meaning p cannot be further expanded so add to the exhausted set of final paths
      add p to set S
    }
  } // End While
  return S
}

S JimpleStatmentsToPathNodes(P) {
  S = {} // Empty set of paths
  for(each path p in set P) {
    pn = [] // Empty list of PathNode
    for(each Jimple statement s in p) {
      // Context c is an object that refers to the origin of method call leading to statement s
      c = context of s
      n = (s,c) // Decorate Jimple statement s with context c
      add n to pn
    }
    add pn to S
  }
  return S
}
```

Figure 3.7: Inter-procedure path generation algorithm.

```
r0 := @this: serl.test.Point [ROOT, FALLS THROUGH]
r1 := @parameter0: java.lang.Object [FALLS THROUGH]
$z0 = virtualinvoke r0.<serl.test.Point: boolean typeCheck(java.lang.Object)>(r1) [FALLS THROUGH]
if $z0 != 0 goto r2 = (serl.test.Point) r1 [FALLS THROUGH]
return 0 [END]
```

Figure 3.8: False returning path due to invalid typeCheck.

```
r0 := @this: serl.test.Point [ROOT, FALLS THROUGH]
r1 := @parameter0: java.lang.Object [FALLS THROUGH]
$z0 = virtualinvoke r0.<serl.test.Point: boolean typeCheck(java.lang.Object)>(r1) [FALLS THROUGH]
if $z0 != 0 goto r2 = (serl.test.Point) r1 [BRANCHES]
r2 = (serl.test.Point) r1 [FALLS THROUGH]
$i0 = r0.<serl.test.Point: int x> [FALLS THROUGH]
$i1 = r2.<serl.test.Point: int x> [FALLS THROUGH]
if $i0 != $i1 goto return 0 [BRANCHES]
return 0 [END]
```

Figure 3.9: False returning path due to unequal x.

for the *typeCheck* method is shown in Figure 3.12. Line 1 and 2 does *this* and *parameter* assignment to $r0$ and $r1$ locals respectively. Line 3 assigns the result of *instanceof* type checking to $z0$. Line 4 checks whether *instanceof* type checking resulted in true in which case line 5 will be executed else line 6 will be executed. If the *Point* class had a subtype that overrode the *typeCheck* method and instead of the *this.typeCheck* call, the *that.typeCheck* was called, then the *class hierarchy analysis* would return two targets for the called method ($Point.typeCheck$ and $SubtypeOfPoint.typeCheck$) and all of the paths from both methods $Point.typeCheck$ and $SubtypeOfPoint.typeCheck$ would have been considered for further processing.

Path generation for Figure 3.12 results in two paths: the first path returns false with failed type checking (Figure 3.13) and the second path returns true with successful type

```
r0 := @this: serl.test.Point [ROOT, FALLS THROUGH]
r1 := @parameter0: java.lang.Object [FALLS THROUGH]
$z0 = virtualinvoke r0.<serl.test.Point: boolean typeCheck(java.lang.Object)>(r1) [FALLS THROUGH]
if $z0 != 0 goto r2 = (serl.test.Point) r1 [BRANCHES]
r2 = (serl.test.Point) r1 [FALLS THROUGH]
$i0 = r0.<serl.test.Point: int x> [FALLS THROUGH]
$i1 = r2.<serl.test.Point: int x> [FALLS THROUGH]
if $i0 != $i1 goto return 0 [FALLS THROUGH]
$i2 = r0.<serl.test.Point: int y> [FALLS THROUGH]
$i3 = r2.<serl.test.Point: int y> [FALLS THROUGH]
if $i2 != $i3 goto return 0 [BRANCHES]
return 0 [END]
```

Figure 3.10: False returning path due to unequal y.

```
1   r0 := @this: serl.test.Point [ROOT, FALLS THROUGH]
2   r1 := @parameter0: java.lang.Object [FALLS THROUGH]
3   $z0 = virtualinvoke r0.<serl.test.Point: boolean typeCheck(java.lang.Object)>(r1) [FALLS THROUGH]
4   if $z0 != 0 goto r2 = (serl.test.Point) r1 [BRANCHES]
5   r2 = (serl.test.Point) r1 [FALLS THROUGH]
6   $i0 = r0.<serl.test.Point: int x> [FALLS THROUGH]
7   $i1 = r2.<serl.test.Point: int x> [FALLS THROUGH]
8   if $i0 != $i1 goto return 0 [FALLS THROUGH]
9   $i2 = r0.<serl.test.Point: int y> [FALLS THROUGH]
10  $i3 = r2.<serl.test.Point: int y> [FALLS THROUGH]
11  if $i2 != $i3 goto return 0 [FALLS THROUGH]
12  return 1 [END]
```

Figure 3.11: True returning path for equal x and y.

```
1  r0 := @this: serl.test.Point
2  r1 := @parameter0: java.lang.Object
3  $z0 = r1 instanceof serl.test.Point
4  if $z0 == 0 goto return 0
5  return 1
6  return 0
```

Figure 3.12: Point.typeCheck(Object) method

checking (Figure 3.14). The algorithm proceeds further where it combines newly generated paths with filtered path. The resulting paths after combination are shown in Figure 3.15 and 3.16. The call site in these paths is replicated to indicate method entry and return point at line number 3 and 9 respectively of both figures. The path in Figure 3.15 is invalid because even though the *typeCheck* method returns false (0 value), the *z0 != 0* check at line number 10 evaluates to true and branches to line number 11. The copy propagation of *BooleanPathFilter* detects this conflict and eliminate this path for containing a dead code block. Any path containing such conflict in boolean value is pruned by *BooleanPathFilter* analysis. As a result only one paths returning true of Figure 3.16 is returned by the path generation module as there is no more method call site to be processed further.

3.4.3 Method Expansion Decision

Optimization is a crucial step in path generation. In absence of a good optimization strategy, paths for the *equals* method may escalate exponentially due to deep call graphs. However, particularly for *equals*, there is some room for optimization through abstraction. Here are some rules for optimization:

```
r0 := @this: serl.test.Point [ENTRY, FALLS THROUGH]
r1 := @parameter0: java.lang.Object [FALLS THROUGH]
$z0 = r1 instanceof serl.test.Point [FALLS THROUGH]
if $z0 == 0 goto return 0 [BRANCHES]
return 0 [RETURN]
```

Figure 3.13: Path for failed type checking

```
r0 := @this: serl.test.Point [ENTRY, FALLS THROUGH]
r1 := @parameter0: java.lang.Object [FALLS THROUGH]
$z0 = r1 instanceof serl.test.Point [FALLS THROUGH]
if $z0 == 0 goto return 0 [FALLS THROUGH]
return 1 [RETURN]
```

Figure 3.14: Path for successful type checking.

Do not expand methods representing state We run pattern detectors on intermediate paths to determine the *this.getM() == that.getM()* pattern that can be translated as *this.m == that.m* in Alloy. Hence the abstraction of the *getM()* method call to the *m* field relaxes the need for path expansion of the *getM()* method.

Do not expand equals or compareTo method on the field We abstract the *this.field.equals(that.field)* or the *this.field.compareTo(that.field) == 0* pattern as two fields being compared for equality i.e *this.field == that.field*. Hence, we do not expand *equals* or *compareTo* method on fields.

Do not expand other methods on the field We do not expand methods on the field (or a method that does not belong to any type in the type hierarchy of the analysis class). Our pattern detector in such cases looks for *this.field.getState() == that.field.getState()* or a similar comparison. This is abstracted as *this.field_state == that.field_state*. This way we transfer a state of a composed field to a state of composing object i.e. *this* or *that*.

Do not expand a static method representing state abstraction Sometimes, certain properties of a field are compared rather than the whole field. This is usually done using static methods. For instance, an equality logic for two integer fields could be evenness or oddness rather than the value of the fields. For instance, *Library.isEven(this.field) == Library.isEven(that.field)* is abstracted as *this.even_field == that.even_field* and

```
1  r0 := @this: serl.test.Point [ROOT, FALLS THROUGH]
2  r1 := @parameter0: java.lang.Object  [FALLS THROUGH]
3  $z0 = virtualinvoke r0.<serl.test.Point: boolean typeCheck(java.lang.Object)>(r1) [FALLS THROUGH]
4  r0 := @this: serl.test.Point [ENTRY, FALLS THROUGH]
5  r1 := @parameter0: java.lang.Object [FALLS THROUGH]
6  $z0 = r1 instanceof serl.test.Point [FALLS THROUGH]
7  if $z0 == 0 goto return 0 [FALLS THROUGH]
8  return 0 [RETURN]
9  $z0 = virtualinvoke r0.<serl.test.Point: boolean typeCheck(java.lang.Object)>(r1) [FALLS THROUGH]
10 if $z0 != 0 goto r2 = (serl.test.Point) r1 [BRANCHES]
11 r2 = (serl.test.Point) r1 [FALLS THROUGH]
12 $i0 = r0.<serl.test.Point: int x> [FALLS THROUGH]
13 $i1 = r2.<serl.test.Point: int x> [FALLS THROUGH]
14 if $i0 != $i1 goto return 0 [FALLS THROUGH]
15 $i2 = r0.<serl.test.Point: int y> [FALLS THROUGH]
16 $i3 = r2.<serl.test.Point: int y> [FALLS THROUGH]
17 if $i2 != $i3 goto return 0 [FALLS THROUGH]
18 return 1 [END]
```

Figure 3.15: Path containing failed type checking.

```
1  r0 := @this: serl.test.Point [ROOT, FALLS THROUGH]
2  r1 := @parameter0: java.lang.Object  [FALLS THROUGH]
3  $z0 = virtualinvoke r0.<serl.test.Point: boolean typeCheck(java.lang.Object)>(r1) [FALLS THROUGH]
4  r0 := @this: serl.test.Point [ENTRY, FALLS THROUGH]
5  r1 := @parameter0: java.lang.Object [FALLS THROUGH]
6  $z0 = r1 instanceof serl.test.Point [FALLS THROUGH]
7  if $z0 == 0 goto return 0 [FALLS THROUGH]
8  return 1 [RETURN]
9  $z0 = virtualinvoke r0.<serl.test.Point: boolean typeCheck(java.lang.Object)>(r1) [FALLS THROUGH]
10 if $z0 != 0 goto r2 = (serl.test.Point) r1 [BRANCHES]
11 r2 = (serl.test.Point) r1 [FALLS THROUGH]
12 $i0 = r0.<serl.test.Point: int x> [FALLS THROUGH]
13 $i1 = r2.<serl.test.Point: int x> [FALLS THROUGH]
14 if $i0 != $i1 goto return 0 [FALLS THROUGH]
15 $i2 = r0.<serl.test.Point: int y> [FALLS THROUGH]
16 $i3 = r2.<serl.test.Point: int y> [FALLS THROUGH]
17 if $i2 != $i3 goto return 0 [FALLS THROUGH]
18 return 1 [END]
```

Figure 3.16: Path containing successful type checking.

the *isEven()* method is not expanded.

Using these rules we save on both space and execution time of *equals* analysis and Alloy code generation. Furthermore, we need smaller number of pattern detectors for handing comparison logic thus providing more code coverage. However, if $f(this.field) = f(that.field) \nRightarrow this.field = that.field$ for an abstraction function f then such abstraction is erroneous and goes undetected. Hence, its a compromise between accuracy and efficiency, and accuracy and code coverage.

3.4.4 Loop Treatment

The *equals* method may not always be as simple as the presented example of the *Point* class. It may have logic for array, list or map comparison, thus, introducing loops within its body. As an approximation, we chose to translate a loop into two non looping paths: one when the control branches to the loop block when the looping condition is true and the other when control falls through when the looping condition is false. To illustrate this, let's look at the ArrayPattern class of Figure 3.17. Line 3 does type checking, line 6 compares the length of the arrays, line 8 initializes, checks and increments the index variable for iteration through the array elements, and line 9 checks for each element in the arrays to be equal. The path generation algorithm produces two paths returning true. The first path comprises line 3,5,6,8 and 12 in which control does not flow inside the looping block (line 9). The second path comprises line 3,5,6,8,9 and 12 in which control flows inside the looping block. Jimple code for the first and second paths are shown in Figure 3.18 and 3.19 respectively. Line 15 of both figures represents a bound check at line 8 ($i < this.array.length$) of Figure 3.17. The first path falls through this check and returns true while the second path branches to the code block where element comparison is done. The second path repeats the same statement at line number 15 and 24 where the condition is considered true at 15 (BRANCHES) and false at line 24 (FALLS THROUGH). Note that we apply a similar approach to recursive calls where it is just expanded once.

```
public class ArrayPattern {
1  private int[] array; // Array field
2  public boolean equals(Object o) {
3    if (!(o instanceof ArrayPattern)) // Type check
4      return false;
5    ArrayPattern that = (ArrayPattern) o;
6    if(this.array.length != that.array.length)  // Size check
7      return false;
8    for(int i = 0; i < this.array.length; ++i) { // Bound check and iteration
9      if(this.array[i] != that.array[i]) // Element check
10      return false;
11   }
12   return true;
13 }
}
```

Figure 3.17: Equals method containing loop for array comparison.

```
1  r0 := @this: serl.test.ArrayPattern [ROOT, FALLS THROUGH]
2  r1 := @parameter0: java.lang.Object [FALLS THROUGH]
3  $z0 = r1 instanceof serl.test.ArrayPattern [FALLS THROUGH]
4  if $z0 != 0 goto r2 = (serl.test.ArrayPattern) r1 [BRANCHES]
5  r2 = (serl.test.ArrayPattern) r1 [FALLS THROUGH]
6  $r3 = r0.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
7  $i1 = lengthof $r3 [FALLS THROUGH]
8  $r4 = r2.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
9  $i2 = lengthof $r4 [FALLS THROUGH]
10 if $i1 == $i2 goto i0 = 0 [BRANCHES]
11 i0 = 0 [FALLS THROUGH]
12 goto [?= $r7 = r0.<serl.test.ArrayPattern: int[] array>] [BRANCHES]
13 $r7 = r0.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
14 $i5 = lengthof $r7 [FALLS THROUGH]
15 if i0 < $i5 goto $r5 = r0.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
16 return 1 [END]
}
```

Figure 3.18: Path in which control does not flow inside the loop block.

3.5 Path, context and flow sensitivities in an analysis

The SERL Control Flow Framework provides a module that accounts for path, context and flow sensitivities in an analysis. The analysis is built upon inter-procedural paths that are traversed statement by statement (Jimple statement) with hooks to several API methods declared as Java APIs in the framework. To illustrate the difference between this approach and regular flow analysis [1], let us look at an example for reaching definition analysis: a simple kind of data-flow analysis that statically evaluates which definition may reach a given point in the code. Figure 3.20 shows an example where variable x at line 5 has a set of reaching definitions x = 0 (line 2) and x = 5 (line 4). We have two reaching definitions at line 5 because of the join of information flowing from two control branches (line 2 and 4) merging at line 5. Two paths can be constructed from this example; the first path consists of lines 1,2,5, ... and the second path consists of lines 1,4,5, ... In the context of the first path x will have value 0 at line 5 and in the context of the second path x will have value 5 at line 5. Path based analysis, in this way, can resolve the ambiguity in the flow analysis due to merging of control branches.

```
1  r0 := @this: serl.test.ArrayPattern [ROOT, FALLS THROUGH]
2  r1 := @parameter0: java.lang.Object [FALLS THROUGH]
3  $z0 = r1 instanceof serl.test.ArrayPattern [FALLS THROUGH]
4  if $z0 != 0 goto r2 = (serl.test.ArrayPattern) r1 [BRANCHES]
5  r2 = (serl.test.ArrayPattern) r1 [FALLS THROUGH]
6  $r3 = r0.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
7  $i1 = lengthof $r3 [FALLS THROUGH]
8  $r4 = r2.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
9  $i2 = lengthof $r4 [FALLS THROUGH]
10 if $i1 == $i2 goto i0 = 0 [BRANCHES]
11 i0 = 0 [FALLS THROUGH]
12 goto [?= $r7 = r0.<serl.test.ArrayPattern: int[] array>] [BRANCHES]
13 $r7 = r0.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
14 $i5 = lengthof $r7 [FALLS THROUGH]
15 if i0 < $i5 goto $r5 = r0.<serl.test.ArrayPattern: int[] array> [BRANCHES]
16 $r5 = r0.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
17 $i3 = $r5[i0] [FALLS THROUGH]
18 $r6 = r2.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
19 $i4 = $r6[i0] [FALLS THROUGH]
20 if $i3 == $i4 goto i0 = i0 + 1 [BRANCHES]
21 i0 = i0 + 1 [FALLS THROUGH]
22 $r7 = r0.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
23 $i5 = lengthof $r7 [FALLS THROUGH]
24 if i0 < $i5 goto $r5 = r0.<serl.test.ArrayPattern: int[] array> [FALLS THROUGH]
25 return 1 [END]
```

Figure 3.19: Path in which control flows inside the loop block.

```
1 if(y < 1)
2   x = 0;
3 else
4   x = 5;
5 if(x < 5)
6 ...
```

Figure 3.20: Reaching Definition Example

3.5.1 Flow Equations for Path Based Approach

Let $gen[S_i]$ be the set of definitions generated at statement S_i, $kill[S_i]$ be the set of definitions killed at S_i, $in[S_i]$ be the set of definitions flowing into S_i and $out[S_i]$ be the set of definitions flowing out of S_i. For the first statement S_1 of a path, equations 3.1 and 3.2 represents the flow of definitions and for S_i where $i > 1$, equations 3.3 and 3.4 represents the flow of definitions in and out of a statement. A path-based approach can eliminate the need for separate *in* flow sets because these equations can be reduced to $out[S_i] = gen[S_i] \cup (out[S_{i-1}] - kill[S_i])$, thus, reducing the memory needed for the analysis by half.

$$in[S_1] = \{\phi\} \tag{3.1}$$

$$out[S_1] = gen[S_1] \tag{3.2}$$

$$in[S_i] = out[S_{i-1}] \tag{3.3}$$

$$out[S_i] = gen[S_i] \cup (in[S_i] - kill[S_i]) \tag{3.4}$$

3.5.2 Context Sensitivity

Inter-procedural flow analysis is context sensitive [2] by nature. As an example let us look at the implementation of the *equals* method of the *ContextExample* class shown in Figure 3.21. There are two true-returning paths produced: one with field *field* of *this* and parameter equal to *null* (Figure 3.22) and the other with *not null* (Figure 3.23). The *getClass()* method has the predefined meaning of computing the runtime type and is not expanded. Similarly, *equals* invocation on a field of an object is also not expanded assuming it has a sound equality implementation.

```
package serl.test;
public class ContextExample {
   private String field;

   public static boolean testNull(Object o) {
     return o == null;
   }

   public boolean equals(Object o) {
     if(testNull(o))
       return false;
     if(this.getClass() != o.getClass())
       return false;
     ContextExample that = (ContextExample)o;
     if(testNull(this.field))
       return testNull(that.field);
     return this.field.equals(that.field);
   }
}
```

Figure 3.21: ContextExample class with mutiple call to testNull static method from equals method.

Figure 3.22 has three call sites for the *testNull* method (line number 3, 14 and 21). For

```
1  r0 := @this: serl.test.ContextExample
2  r1 := @parameter0: java.lang.Object
3  $z0 = staticinvoke <serl.test.ContextExample: boolean testNull(java.lang.Object)>(r1)
4  r0 := @parameter0: java.lang.Object
5  if r0 != null goto return 0
6  return 0
7  $z0 = staticinvoke <serl.test.ContextExample: boolean testNull(java.lang.Object)>(r1)
8  if $z0 == 0 goto $r3 = virtualinvoke r0.<java.lang.Object: java.lang.Class getClass()>()
9  $r3 = virtualinvoke r0.<java.lang.Object: java.lang.Class getClass()>()
10 $r4 = virtualinvoke r1.<java.lang.Object: java.lang.Class getClass()>()
11 if $r3 == $r4 goto r2 = (serl.test.ContextExample) r1
12 r2 = (serl.test.ContextExample) r1
13 $r5 = r0.<serl.test.ContextExample: java.lang.String field>
14 $z1 = staticinvoke <serl.test.ContextExample: boolean testNull(java.lang.Object)>($r5)
15 r0 := @parameter0: java.lang.Object
16 if r0 != null goto return 0
17 return 1
18 $z1 = staticinvoke <serl.test.ContextExample: boolean testNull(java.lang.Object)>($r5)
19 if $z1 == 0 goto $r7 = r0.<serl.test.ContextExample: java.lang.String field>
20 $r6 = r2.<serl.test.ContextExample: java.lang.String field>
21 $z2 = staticinvoke <serl.test.ContextExample: boolean testNull(java.lang.Object)>($r6)
22 r0 := @parameter0: java.lang.Object
23 if r0 != null goto return 0
24 return 1
25 $z2 = staticinvoke <serl.test.ContextExample: boolean testNull(java.lang.Object)>($r6)
26 return $z2
```

Figure 3.22: ContextExample.equals with this.field and that.field null.

context sensitivity, the path generator has cloned the path for different call sites. Even though the block containing line numbers 15-17 and 22-24 are the same, they are not equal as these paths are results of different calling contexts (14 and 21 respectively). Similarly, local variables with the name *r0* declared at line 1, 4, 15, and 22 are not equal even though they share same name because they have different calling contexts. These facts also apply to Figure 3.23. Path blocks 4-6 and 15-17 are considered unequal even though they are the same set of statements generated from path generation of the *typeCheck* method. Similarly, identically named variables at different calling contexts are not equal in this path as well.

Context sensitivity is achieved by decorating each Jimple statement with a context object. The decorated Jimple statement is called *PathNode*. Two different PathNodes in a path are not equal; they have different physical memory. A context object C is of a form: C = *(call site, resolved method, resolved class)*. It contains a call site (a PathNode linking back to the calling statement), a resolved method (method that is resolved using CHA for given invoke expression at the call site), and a resolved class (a class containing the resolved method). It is possible to find the declaring class of a method directly using

```
1  r0 := @this: serl.test.ContextExample
2  r1 := @parameter0: java.lang.Object
3  $z0 = staticinvoke <serl.test.ContextExample: boolean testNull(java.lang.Object)>(r1)
4  r0 := @parameter0: java.lang.Object
5  if r0 != null goto return 0
6  return 0
7  $z0 = staticinvoke <serl.test.ContextExample: boolean testNull(java.lang.Object)>(r1)
8  if $z0 == 0 goto $r3 = virtualinvoke r0.<java.lang.Object: java.lang.Class getClass()>()
9  $r3 = virtualinvoke r0.<java.lang.Object: java.lang.Class getClass()>()
10 $r4 = virtualinvoke r1.<java.lang.Object: java.lang.Class getClass()>()
11 if $r3 == $r4 goto r2 = (serl.test.ContextExample) r1
12 r2 = (serl.test.ContextExample) r1
13 $r5 = r0.<serl.test.ContextExample: java.lang.String field>
14 $z1 = staticinvoke <serl.test.ContextExample: boolean testNull(java.lang.Object)>($r5)
15 r0 := @parameter0: java.lang.Object
16 if r0 != null goto return 0
17 return 0
18 $z1 = staticinvoke <serl.test.ContextExample: boolean testNull(java.lang.Object)>($r5)
19 if $z1 == 0 goto $r7 = r0.<serl.test.ContextExample: java.lang.String field>
20 $r7 = r0.<serl.test.ContextExample: java.lang.String field>
21 $r8 = r2.<serl.test.ContextExample: java.lang.String field>
22 $z3 = virtualinvoke $r7.<java.lang.String: boolean equals(java.lang.Object)>($r8)
23 return $z3
```

Figure 3.23: ContextExample.equals with this.field and that.field not null.

the given method of context object. However, at times, a subclass will not override some of the parent methods and if analysis of a parent method is to be done in the context of the subclass then class information is needed for a context object. Let us represent each statement of Figure 3.23 by S_1, S_2, ..., S_{23}. Altogether, the path requires three context objects to model context sensitivity:

- Statements $\{S_{1-3}, S_{7-14}, S_{18-23}\}$ are assigned a context $C_1 = (null, equals,$ $ContextExample)$. A null call site in C_1 implies that there is no calling site for the *equals* method of the *ContextExample* class as this is the entry method. C_1 is applied to all of the statements in the *equals* method excluding statements resulted from processing of invoke expressions.

- Statements S_{4-6} are assigned a context $C_2 = (S_3, testNull, ContextExample)$. The statement block S_{4-6} was produced from the invoke expression at statement S_3 which was resolved to the *testNull* method of the *ContextExample* class.

- Statements S_{15-17} are assigned a context $C_3 = (S_{14}, testNull, ContextExample)$. The statement block S_{15-17} was produced from the invoke expression at statement S_{14} which was resolved to the *testNull* method of the *ContextExample* class.

A *PathNode* representing a Jimple statement with context as decorating object only establishes statement level context sensitivity. The underlying local variables/references of two identical Jimple statements taken from the same method but expanded at different points in a path share the same memory location essentially making them equal. For example, in Figure 3.23, local variable $r0$ at line 4 is equal to $r0$ at line 15 as both of these variables and corresponding Jimple statements came from the same method $testNull$ that maintains a single copy of its statements in the memory. Hence, a variable/reference level context sensitivity is also required to distinguish $r0$ at line 4 from $r0$ at line 15. This is done by another decorator object called $ValueInContext$ of the form $V = (LocalVariable, Context)$. Local variable is a Jimple object for a local variable like $r0$ and context is the context of the *PathNode* where this local variable is defined or used. Hence, $r0$ at line 4 is represented as $V_1 = (r0, C_2)$ and $r0$ at line 15 is represented as $V_2 = (r0, C_3)$, C_1 and C_2 being the context of line 4 and 15 respectively. V_1 and V_2 are now two different entities due to difference in contexts. In the presence of reference sensitivity, a flow analysis will not overwrite the flow set for r_0 at line 4 with the flow set for $r0$ at line 15 and it will be possible to perform interprocedural flow analysis like copy propagation analysis that can map back an assignments to a local variable at a called method to the locals or globals declared or used at the calling method.

3.6 Code Pattern Detection

A *path* produced by the path generator is essentially a list of *PathNodes* (containing Jimple statements) with their execution order preserved. The code pattern detector tries to query the structure as well as execution order of these statements to extract some pattern in the paths being analyzed. A specialized copy propagation analysis tool called ReferencePropagationAnalysis is built to run on top of these paths to extract two kinds of information: copy root and composing root. Both of these analyses are repeatedly used for identification of code patterns in a path.

3.6.1 Copy Root

The Copy root of a variable is the root assignment of that variable in a given path. The Copy root of a variable at a point (before or after a statement) in a path is evaluated with the following recursive rules:

1. CopyRoot(Expression) = Expression, e.g. CopyRoot(this.field) = this.field, CopyRoot(this.equals(that)) = this.equals(that), CopyRoot(this.field[i]) = this.field[i] or CopyRoot(a-b) = a - b.

2. CopyRoot(Value) = CopyRoot(Expression), if the path has a definition statement such as "Value = Expression" provided this definition is not killed before the evaluation point, e.g. immediately after "x = this.field" statement, CopyRoot(x) = CopyRoot(this.field) = this.field. Similarly, for path [x = this; y = x; return y;], CopyRoot(y) = this after "y = x;" statement.

3. CopyRoot(Formal Parameter) = Formal Parameter, for entry method.

4. CopyRoot(Formal Parameter) = CopyRoot(Actual Parameter), for non-entry method.

3.6.2 Composing Root

The Composing root of a reference variable is the *base* of the root dereferencing assignment in a path. The Composing root of a reference variable at a point (before or after a statement) in a path is evaluated with following rules:

1. ComposingRoot(Non Object Reference/Value) = null, e.g. ComposingRoot(100) = null.

2. ComposingRoot(Non Object Evaluating Expression) = null, e.g. ComposingRoot(a - b) = null.

3. ComposingRoot(This Reference) = This Reference, e.g. ComposingRoot(this) = this.

4. ComposingRoot(Formal Parameter) = Formal Parameter, for entry method.

5. ComposingRoot(Formal Parameter) = ComposingRoot(Actual Parameter), for non entry method.

6. ComposingRoot(Static Dereferencing Expression) = null, e.g. ComposingRoot(Type.field) = null, ComposingRoot(Type.method(param)) = null.

7. ComposingRoot(Pointer Dereferencing Expression) = ComposingRoot(Pointer), e.g. ComposingRoot(this.field) = ComposingRoot(this) = this, ComposingRoot(this.equals(that)) = ComposingRoot(this) = this, ComposingRoot(this.field[i]) = ComposingRoot(this.field) = ComposingRoot(this) = this.

8. ComposingRoot(Value) = ComposingRoot(Expression), if the path has a definition statement such as "Value = Expression" provided this definition is not killed before the evaluation point e.g. for path [x = this; y = x; return y;], ComposingRoot(y) = ComposingRoot(x) = ComposingRoot(this) = this after"y = x;" statement which is same as copy root. Similarly, for path [a = this; b = a; c = b.field; d = c; return d;], ComposingRoot(d) = ComposingRoot(c) = ComposingRoot(b.field) = ComposingRoot(b) = ComposingRoot(a) = ComposingRoot(this) = this, after "d = c;" statement.

Let's consider the path in Figure 3.23. At line 2, the copy root of *r1* is *parameter0* (or *Object o*). At line 4 the copy root of *r0* is *parameter0* of the *equals* method (not *parameter0* of *typeCheck*) because *r1* of the *equals* method is assigned to *r0* of *typeCheck* due to aliasing of the actual parameter (argument) and formal parameter. Similarly, at line 20, copy root of *r7* is *r0.field*, a field reference expression and copy root of *z3* at line 23 is *r7.equals(r8)*, a virtual invoke expression. The Composing root of *r0* is the *this* reference at line 1. The Composing root of *r5* at line number 13 is the *this* reference and the composing root of *r8* at line number 21 is *parameter0* (or *Object o*).

The copy root and composing root are very important constructs for Jimple code pattern detection. The detected patterns are used for error reporting and association with alloy code. The pattern detector creates a mapping of one or more Jimple statement to an alloy code element which is later translated to a full alloy module by *CodeGenerator* discussed in

Section 3.7. Some of the interesting code patterns detected are explained in the following sections.

3.6.3 Type Checking Pattern

Type checking in Java can be done in several ways. The pattern detector detects four different kinds of type checking code patterns. Assuming a *Point* class and parameter *o* of the *equals* method, type checking can be done in the following ways:

1. Using *instanceof* operator, e.g. o instanceof Point
2. Using *getClass()* method, e.g. this.getClass() == o.getClass()
3. Using *Type.class* static field, e.g. o.getClass() == Point.class
4. Using *ClassCastException*, e.g. try{Point that = (Point)o; . . . }catch(ClassCastException e){return false;}

Here we present an example of a detector that detects *instanceof* type checking for the *equals* method. The algorithm shown in Figure 3.24 takes a path P as input and outputs a set of nodes S containing *instanceof* type checking patterns in P. Let us run the algorithm in the *equals* method of a *Person* class shown in Figure 3.25. But first, we need to apply the inter-procedure path generator to get corresponding paths for the *equals* method; in this case, we just get 1 as shown in Figure 3.26 . The algorithm works in the following way:

1. Iterates through Jimple statements in the path looking for an *IfStmt*. It finds one at line 4.
2. Gets conditional expression (with *== or !=*) of *if* statement and check if right operand is integer constant or not. In this case, *$z0 != 0*, and is true.
3. Checks if copy root of left local variable is *instanceof* expression or not. *CopyRoot($z0) = (r1 instanceof serl.test.Person)*, which is an *instanceof* expression.

```
S InstanceOfTypeCheckDetector(P) {
  S = {} // Set of PathNode containing type check statment
  i = 0 // Sentinel for iterating over P
  for(i = 0 to length(P) - 1) {
    s = statement of path node P[i]
    if(s is If statment) {
      c = condition expression of s
      if(c has == or != conditional operator) {
        rOp = right operand of c
        if(rOp is integer constant) {
          lOp = left operand of c
          cpLOp = CopyRoot(lOp)
          if(cpLOp is instance of check expression) {
            iOp = operand of cpLOp
            cpIOp = CopyRoot(iOp)
            if(cpIOp is first parameter) {
              add P[i] to S
            }
          }
        }
      }
    }
  }
  return S
}
```

Figure 3.24: Detector for instanceof type checking pattern.

```
package serl.test;
public class Person {
1  private int age;
2  private String name;

3  public boolean equals(Object o) {
4    if(!(o instanceof Person))
5      return false;
6    Person that = (Person) o;
7    return this.age == that.age &&
8      this.name.equals(that.name);
9  }
}
```

Figure 3.25: Person.equals method with instanceof type checking.

```
1  r0 := @this: serl.test.Person
2  r1 := @parameter0: java.lang.Object
3  $z0 = r1 instanceof serl.test.Person
4  if $z0 != 0 goto r2 = (serl.test.Person) r1
5  r2 = (serl.test.Person) r1
6  $i0 = r0.<serl.test.Person: int age>
7  $i1 = r2.<serl.test.Person: int age>
8  if $i0 != $i1 goto return 0
9  $r3 = r0.<serl.test.Person: java.lang.String name>
10 $r4 = r2.<serl.test.Person: java.lang.String name>
11 $z1 = virtualinvoke $r3.<java.lang.String: boolean equals(java.lang.Object)>($r4)
12 if $z1 == 0 goto return 0
13 return 1
```

Figure 3.26: Path for Point.equals method of Figure 3.25.

4. Checks if copy root of operand of *instanceof* expression is *parameter0* or not. *CopyRoot(r1) = parameter0* in this case. Thus, a type checking statement has been identified and added to *S*.

5. This process repeats until all of the statements in the paths are traversed.

3.6.4 State Equality Pattern

A state of an object can be a value in a field or a value returned by a getter method or can be a method itself that was not expanded (e.g. interface method like *java.util.Collection.size()* in Java which has a pre-defined meaning of returning the size of a collection). The pattern detector for state checking pattern works in two ways: one for equality comparison using the == or *!*= operator and the other for equality comparison using the *equals* or *compareTo* method. Let's consider the path in Figure 3.26 to understand the internal working of the state check detector shown in Figure 3.27.

1. It Iterates through Jimple statements in the path looking for an *IfStmt*. It finds one at line 4.

2. Checks if copy roots of both left and right operands are field references. This is not true.

3. Checks if copy root of left operand is invoke expression and right operand is integer constant. Copy root of left operand is *instanceof* invoke expression. So, this is not true.

4. Checks if copy root of left operand is == or *!*= conditional expression and right operand is integer. This is also not true.

5. It iterates further till it reaches line 8 where it finds *IfStmt*.

6. Checks if copy roots of both left and right operands are field reference. This is true (*ro.age* and *r2.age*).

```
S StateCheckDetector(P) {
  S = {} // Set of PathNode containing state check statment
  i = 0 // Sentinel for iterating over P
  for(i = 0 to length(P) - 1) {
    s = statement of path node P[i]
    if(s is If statment) {
      c = condition expression of s
      if(c has == or != conditional operator) {
        lOp = left operand of c
        rOp = right operand of c
        lCpOp = CopyRoot(lOp)
        rCpOp = CopyRoot(rOp)

        // Note instead of field reference an unexpanded
        // instance invoke expression could be present
        if(lCpOp is field reference && rCpOp is field reference) {
          if(CheckThisThat(lOp, rOp)) {
            // Detected pattern: if(this.field == that.field) ...
            add P[i] to S
          }
        }
        else if(lCpOp is instance invoke expression && rOp is integer constant) {
          m = method in lCpOp
          if(m is equals method || m is compareTo method) {
            // m could be of form b.equals(a) or b.compareTo(a)
            b = base of lCpOp
            a = arguement(0) of lCpOp
            aCp = CopyRoot(a)
            bCp = CopyRoot(b)

            if(aCp is field reference && bCp is field reference) {
              if(CheckThisThat(a, b)) {
                // Detected pattern: if(this.field.equals(that.field)) ...
                add P[i] to S
              }
            }
          }
        }
        else if(lCpOp is == or != conditional expression && rOp is integer constant) {
          nLOp = left operand of lCpOp
          nROp = right operand of lCpOp
          nLCpOp = CopyRoot(lOp)
          nRCpOp = CopyRoot(rOp)

          if(nLCpOp is field reference && nRCpOp is field reference) {
            if(CheckThisThat(nLOp, nROp)) {
              // Detected pattern: [temp = (this.field == that.field); if(temp == 1) ...]
              add P[i] to S
            }
          }
        }
      }
    }
  }
  return S
}

// Checks wheter composing root of lOp and rOp are this reference and parameter(0)
// reference respectively or vice versa; i.e. this and o for equals(Object o) method
boolean CheckThisThat(lOp, rOp) {
  lCmOp = ComposingRoot(lOp)
  rCmOp = ComposingRoot(rOp)

  if((lCmOp is this reference && rCmOp is parameter(0) reference) ||
  (rCmOp is this reference && lCmOp is parameter(0) reference) ) {
    return true
  }
  return false
}
```

Figure 3.27: Detector for instanceof type checking pattern.

7. It then calls *CheckThisThat(i0, i1)* to check if composing root of *i0* and *i1* are *this* and *parameter* reference respectively or vice versa. This is true as *ComposingRoot(i0) = @this* and *ComposingRoot(i1) = @parameter0*.

8. We have detected *this.age == that.age* pattern, so, the algorithm adds statement at line 8 to *S*.

9. It further iterates through statements until its reaches line 12.

10. This time the *else if* block where copy root of left operand is *instance invoke expression*[3] is satisfied.

11. Base of invoke expression is *r2* and 0^{th} argument is *$r4*.

12. The algorithm checks if copy root of base *$r3* and argument *$r4* are field references. True in this case: *CopyRoot($r3) = r0.name* and *CopyRoot($r4) = r2.name*.

13. Checks if the composing roots of *$r3* and *$r4* are *this* and *parameter0* respectively or vice versa. True is this case: *ComposingRoot($r3) = @this* and *ComposingRoot($r4) = @parameter0*.

14. This time we have detected *this.name.equals(that.name)* check pattern and statement at line 12 is added to *S*.

15. The algorithm then iterates through rest of the path checking for these conditions. After the algorithm terminates S will have statement at line 8 and 12 as state checking statement.

Similarly, there are more detectors for checking array equality patterns, Java collection's List, Map and Set equality pattern have been developed but instead of presenting algorithms for them which tend to get longer, we present example source code and conditions that we look for in order to classify a set of PathNodes as a pattern. Particularly, three patterns will be discussed: Array, Set and Map equality pattern.

[3]Note that *InstanceInvokeExpr* in Soot is super type of *VirtualInvokeExpr* and *InterfaceInvokeExpr* meaning these expression dereference object pointers not static class pointers which is done by *StaticInvokeExpr*.

3.6.5 Array Equality Pattern

In array equality pattern we look for three different components in the source code: *size check*, *bound check* and *elements check*. Let us re-consider the *ArrayPattern* class shown in Figure 3.17. Size check enforces the size of two arrays (*this.array* and *that.array*) to be equal, bound check enforces that the same sentinel is used to iterate through *this.array* and *that.array* and do not exceed size of these arrays. Element check ensures that each element in the two arrays are also equal.

Size check in the figure is done using *if(this.array.length != that.array.length) return false;* statements. Nevertheless, it can also be done using *if(this.array.length-that.array.length <= 0) return false;*. There can be several ways to write this using several combinations of variables whose value can be known only at runtime. To generalize, we abstract out the underlying details of the implementation and look for a conditional expression that uses *this.array.length* and *that.array.length* as operands and evaluates to true or false. This may introduce false positives like $if(this.array.length - that.array.length == 1)$ as valid size check which is wrong. However, for the sake of generalization we will accept such a check as a valid size check. Hence, a size check pattern is an abstract boolean function s such that $y_1 = s(this.array.length, that.array.length)$ where $y_1 = (\top \vee \bot)$.

Bound check tries to enforce the array iterator to not exceed the size of arrays being compared. It is done in line 8 of Figure 3.17 as *for(int i = 0; i < this.array.length; ++i)*. Like size check, bound check is also an abstract boolean function b that takes $i \in integer_+$ and one of the arrays lengths as parameter such that $y_2 = b(i, this.array.length \vee that.array.length)$ where $y_2 = (\top \vee \bot)$ provided $y_1 = \top$ i.e. the bound check must follow from size check.

The third check for array pattern is element check, where we iterate through each element of both arrays and compare them to be equal. In Figure 3.17, this is done at line 10 as *if(this.array[i] != that.array[i]) return false*. The equality check of an array element accessed in part by bound variable i is an abstract boolean function e such that $y_3 = e(this.array[i + \alpha], that.array[i + \beta])$ for some integer α and β where $y_3 = (\top \vee \bot)$

provided both $y_1 = \top$ and $y_2 = \top$.

Note that there are two paths (Figure 3.18 and 3.19) generated for the *ArrayPattern* class of Figure 3.17. Figure 3.18 represents a path where size check and bound check is done but the bound condition is not satisfied. This path is valid for empty arrays or when the loop sentinel reaches the array bound. Figure 3.19 represents a path where all three checks are done. This path is valid for non-empty arrays where the loop sentinel has not reached the bound yet. We can combine these two paths and abstract equality comparison of two array by a simple statement *this.array == that.array* in Alloy where *array* is an Alloy sequence. Note that this abstraction is just an approximation and sometimes may hide array related errors from being detected. Nevertheless, our analysis of several real life projects show that the abstraction works for the majority of the cases as shown in Section 4.4, Table 4.3.

Array equality pattern and *List equality pattern* (for subtypes of *java.util.List* of Java's collection framework) are almost identical. The only difference is instead of using an integer sentinel (in array pattern), list uses either *java.util.Iterator* or *java.util.ListIterator* to iterate through their elements in the *equals* method.

3.6.6 Set Equality Pattern

In the Set equality pattern we look for two components in the source code: size check and containment check. An *equals* method of *java.util.AbstractSet* is shown in Figure 3.28. Size check is done through *if(c.size != size()) return false* and containment check is done through a *return containsAll(c)* statement. Instead of size check there could be an inefficient bidirectional containment check *this.containsAll(c) && c.containsAll(this)* which is not considered by the checker as a proper set pattern as the previous approach is more efficient. If we find size and a containment check in a path then we conclude two sets are being compared. Assuming an abstract state *set* as a member field for the *AbstractSet* class and *that* as an object the *this* set is compared against, the logic for set equality can be abstracted as:

```
public boolean equals(Object o) {
  // Reference check
  if (o == this)
    return true;

  // Type check
  if (!(o instanceof Set))
    return false;

  // Set is subtype of Collection, that specifies containsAll method
  Collection c = (Collection) o;

  // Size check
  if (c.size() != size())
    return false;

  try {
    // Containment check
    return containsAll(c);
  } catch(ClassCastException unused)   {
    return false;
  } catch(NullPointerException unused) {
    return false;
  }
}
```

Figure 3.28: Equals method of java.util.AbstractSet.

$$[this.size() = that.size() \wedge$$
$$this.containsAll(that)]$$
$$\vdash (this.set = that.set) \tag{3.5}$$

Note that the correctness of this abstraction also depends on how the *size* and *containAll* method are implemented. These methods are interface methods and serve as a standard specification with predefined meaning. Hence, we do not expand these interface methods. If these methods are implemented incorrectly then such errors will go undetected by the checker. Ideally, we are assuming that these methods are implemented to give the following abstraction:

$$[|this.set| = |that.set| \wedge$$
$$\forall_e (e \in this.set \Rightarrow e \in that.set)]$$
$$\vdash (this.set = that.set) \tag{3.6}$$

3.6.7 Map Equality Pattern

In the Map equality pattern we look for three components: size check, bound check and map element check like in the array equality pattern. In *equals* method of the *java.util.AbstractMap* class of Figure 3.29, the size check is done in line 9 and 10, the bound check for iterator is done in line 16 and the element check is done in line 22-29 where it first checks for an equal key to null mapping in both maps in line 23-24 or an equal key to equal value mapping in line 26-27. Like in the set, we are assuming that all of the unexpanded interface methods are implemented according to the specification of the Java collection framework. If these methods do not behave as specified by the specification then such cases will go undetected by the checker. Assuming an abstract *map* containing set of key to value mapping pairs, (k, v), as member field for *AbstractMap* class and comparison of the *this* map against the *that* map object, the logic for the map equality pattern can be abstracted as follows:

$$\begin{aligned} &[|this.map| = |that.map| \wedge \\ &\forall_{(k,v)}((k,v) \in this.map \Rightarrow (k,v) \in that.map)] \\ &\vdash (this.map = that.map) \end{aligned} \tag{3.7}$$

Ideally, we are checking that *this.map* and *that.map* have same key to value mapping.

3.7 Alloy Code Generation

Alloy code generation process relies on Eclipse's JDT for type hierarchy computation and Serl's control-flow framework for path generation and pattern detection. Each detected pattern is converted to corresponding alloy statements. An alloy module (a compilable Alloy file) consists of logical abstraction of *equals* method implementations and a specification to check the correctness of the implementations. Each module comprises abstraction of classes in a type hierarchy that override or inherit the *equals* method. The language specification for the alloy model can be found in Appendix A.

Here, we will look at important components of alloy model with reference to the *equals*

```
public boolean equals(Object o) {
1   // Reference check
2   if (o == this)
3     return true;

4   // Type Check
5   if (!(o instanceof Map))
6     return false;

7   Map<K,V> t = (Map<K,V>) o;

8   // Size Check
9   if (t.size() != size())
10    return false;

11  try {
12    // Iterates through each mapping(key -> value) of this map
13    // and checks for equal mapping in t map
14    Iterator<Entry<K,V>> i = entrySet().iterator();

15    // Bound Check
16    while (i.hasNext()) {
17      Entry<K,V> e = i.next();
18      K key = e.getKey();
19      V value = e.getValue();

20      // Eement check: checks if for all equal key we have
21      // equal value for both null or not null value cases
22      if (value == null) {
23        if (!(t.get(key)==null && t.containsKey(key)))
24          return false;
25      } else {
26          if (!value.equals(t.get(key)))
27            return false;
28      }
29    }
30  } catch(ClassCastException unused) {
31    return false;
32  } catch(NullPointerException unused) {
33    return false;
34  }
35  return true;
}
```

Figure 3.29: Equals method of java.util.AbstractMap.

method of the *FieldAddedMap* class shown in Figure 3.30. The type hierarchy for *FieldAddedMap* is shown in Figure 3.31. Here *AbstractMap* and *FieldAddedMap* override the *equals* method of the *Object* class. Classes implementing the *equals* method are marked with *. Line 3 of Figure 3.30 has a *super.equals* call which is delegated to *HashMap*. *HashMap*, however, does not override the *equals* method of *AbstractMap* and thus this call is resolved to the *equals* method of *AbstractMap* that was shown in Figure 3.29.

The generated alloy module for type hierarchy involving the *FieldAddedMap* class is shown in Figure 3.32. The six top level blocks of the module are discussed in the following

```
public class FieldAddedMap<K, V> extends HashMap<K, V> {
1  private char field;

2  public boolean equals(Object o) {
3    if(!super.equals(o))
4      return false;

5    if(!(o instanceof FieldAddedMap))
6      return false;

7    FieldAddedMap<K,V> that = (FieldAddedMap<K,V>)o;
8    return this.field == that.field;
9  }
}
```

Figure 3.30: FieldAddedMap implementing equals method.

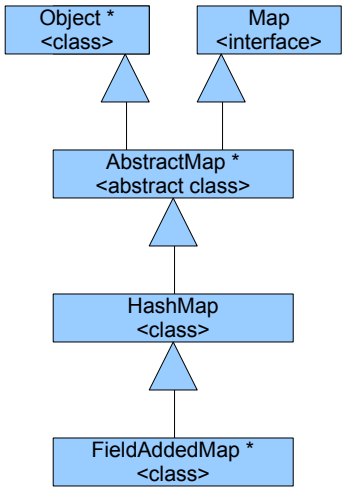


Figure 3.31: Type hierarchy involving FieldAddMap class.

sections.

3.7.1 Module Declaration

The module declaration defines the name of the module. Each module is associated with a type hierarchy. The name in the figure: *analysis/alloy/java_util_AbstractMap_Hierarchy* means that this file can be found in directory *analysis/alloy/* and the name *java_util_AbstractMap_Hierarchy* means that the type hierarchy is computed for *java.util.AbstractMap* in the context of the project being analyzed. This class is the first class down a type hierarchy that overrides the *equals* method of the *java.lang.Object* class.

```
//**** Module declaration Block ****//
module analysis/alloy/java_util_AbstractMap_Hierarchy // AbstractMap is the base type

//**** Type Declaration Block ****//
abstract sig java_lang_Object {} // java.lang.Object class declaration

sig java_util_Map in java_lang_Object { // java.util.Map interface declaration
   java_util_Map_map : Int -> Int       // All maping are abstracted as int to int
}

abstract sig java_util_AbstractMap extends java_lang_Object {} // abstract for abstract Java type
fact { java_util_AbstractMap in java_util_Map }            // AbstractMap implements Map interface

sig java_util_HashMap extends java_util_AbstractMap {} // HashMap inherits AbstractMap

sig serl_map_FieldAddedMap extends java_util_HashMap { // FieldAddedMap inherits HashMap
   serl_map_FieldAddedMap_field : Int                  // All fields are abstracted as Int
}

//**** Equality Predicate Declaration Block ****//
pred java_lang_Object :: equals( that: java_lang_Object ) {
   ( this in java_util_HashMap -  serl_map_FieldAddedMap) =>  // HashMap implementation
   (  // Start of Implication Block
      ( this = that ) // A Fact
      or
      ( // Start of a FactBlock
         !( this = that ) and ( that in java_util_Map ) and
         ( this.java_util_Map_map = that.java_util_Map_map )
      )
   )
   else
   ( this in serl_map_FieldAddedMap ) => // FieldAddedMap Implementation
   (
      (
         ( this = that ) and ( that in serl_map_FieldAddedMap ) and
         ( this.serl_map_FieldAddedMap_field = that.serl_map_FieldAddedMap_field )
      )
      or
      (
         !( this = that ) and ( that in java_util_Map ) and
         ( this.java_util_Map_map = that.java_util_Map_map ) and
         ( that in serl_map_FieldAddedMap ) and
         ( this.serl_map_FieldAddedMap_field = that.serl_map_FieldAddedMap_field )
      )
   )
}

//**** Type Exclusion Specification Block ****//
sig ExclusionSet in java_lang_Object {}
fact {
 java_util_Map -  java_util_AbstractMap - java_util_HashMap -  serl_map_FieldAddedMap
 in ExclusionSet
}
fact {
 java_util_AbstractMap - java_util_HashMap -  serl_map_FieldAddedMap in ExclusionSet
}

//**** Equality Properties Specification Block ****//
assert reflexive { all a : java_lang_Object - ExclusionSet | a.equals[a] }
assert symmetric { all a, b: java_lang_Object - ExclusionSet | a.equals[b] <=> b.equals[a] }
assert transitive {
  all a, b, c: java_lang_Object - ExclusionSet | a.equals[b] and b.equals[c] => a.equals[c]
}

//**** Validity Check Block ****//
// Checks implementation vs specification
check reflexive for 1 // Check for 1 object
check symmetric for 2 // Check for 2 objects
check transitive for 3 // Check for 3 objects
```

Figure 3.32: Alloy module for type hierarchy involving FieldAddedMap.

3.7.2 Type Declaration

The type declaration block defines all of the associated types in a type hierarchy. There are two components of the type declaration: *field* specification and *sig* (Alloy Signature) specification.

Field Specification

The rules for field specification are as follow:

1. All of the fields are abstracted as *Int* type. In Figure 3.32, the *char* field of *Field-AddedMap* is abstracted as *Int* field.

2. A method invocation where the receiver is a field is abstracted as a new field that contains name of both the field and the method. For example, if class T with field f has a *this.f.m() == that.f.m()* check in the *equals* method then the field of *T* will be *T_f_m* not *T_f*. The method *m()* is not expanded, in fact, only methods of non composed types are expanded provided they belong to the type hierarchy of the type being analyzed. Similarly, for *this.f1.m1().f2.m2() == that.f1.m1().f2.m2()* check, the field name used is: *T_f1_m1_f2_m2*.

3. An array field is declared as an Alloy sequence. For example, for type *T* with array field *f[]*, the alloy representation will be: *T_f : seq Int*.

4. For a type inheriting from *java.util.Map*, there are several methods/fields involved in equality checking. However we abstract the implementation details of map by introducing a map Alloy field *java_util_Map_map : Int -> Int* to the type *java_util_Map*. All of the paths of the *equals* method of a *Map* type class are evaluated to match abstraction logic (Relation 3.7) of Section 3.6.7.

5. For a type inheriting from *java.util.Set*, like in map we abstract the implementation details of set by introducing a set Alloy field *java_util_Set_set : set Int* to the type

java_util_Set. All of the paths of the *equals* method of the *Set* type class are evaluated to match abstraction logic (Relation 3.6) of Section 3.6.6.

6. Similarly for a type inheriting from *java.util.List*, we abstract the implementation details by introducing an Alloy sequence field *java_util_List_list : seq Int* to the type *java_util_List*. All of the paths of the *equals* method of the *List* type class are evaluated to match array abstraction logic of Section 3.6.5.

Sig Specification

The rules for sig specification are as follow:

1. An abstract class in Java is declared as *abstract sig* in Alloy. Interfaces and regular classes are declared only *sig*.

2. In Java, *java.lang.Object* is parent type of all the types. We declare it as an abstract type because it does not contribute to the logic of equality comparison for subtypes. All of the alloy sig inherits from *java_lang_Object* sig.

3. Java supports multiple inheritance through interface. A type in java can extend only one type but can implement multiple interface types. Similarly in alloy, a sig can extend one type but we use set containment operator *in* within a fact block to specify interface inheritance. As a rule, for a Java class, we use *extends* keyword of Alloy to model class inheritance and *in* keyword to model interface inheritance (e.g. *abstract sig java_util_AbstractMap extends java_lang_Object* {} *fact* { *java_util_AbstractMap in java_util_Map* }). However, for a Java interface, we use *in* keyword all the time to model inheritance (e.g. *sig java_util_Map in java_lang_Object* { *java_util_Map_map : Int -> Int* }).

3.7.3 Equality Predicate

Equality predicate defines the implementation logic of all of the *equals* method in a type hierarchy. This predicate is defined as a predicate of *java_lang_Object* that takes *that*

(also of type *java_lang_Object*) as a parameter. The equality predicate is composed of several blocks called *implication blocks* separated by *else*. (e.g. *(this in java_util_HashMap - serl_map_FieldAddedMap) => (. . .) else (this in serl_map_FieldAddedMap) => (. . .)*). An *implication block* is further composed of *fact blocks* (separated by *or*) which is composed of logical statements called *facts*. A *fact* can be *type checking* (e.g. *that in serl_map_FieldAddedMap*) or *state checking* fact (e.g. *this.field = that.field*). We stop at this level of object modeling.

An implication block is associated with a *concrete* (*non-abstract*, *non-interface*) type. It represents the implementation of the *equals* method for that type. There could be multiple paths generated for the *equals* method. Sometimes, these paths are abstracted as single *fact block* (we call it *Reduction Translation*) while other times they are translated to multiple *fact blocks* (we call it *One-to-One Translation*).

Reduction Translation

If we consider the equality implementation of *AbstractMap* shown in Figure 3.29, we get four true-returning paths:

```
a) 1,2,3
-------------------------------------------------------------------------------
b) 1,2,4,5,7,8,9,11,12,13,14,15,16,35
-------------------------------------------------------------------------------
c) 1,2,4,5,7,8,9,11,12,13,14,15,16,17,18,19,20,21,22,23,16,35
-------------------------------------------------------------------------------
d) 1,2,4,5,7,8,9,11,12,13,14,15,16,17,18,19,20,21,22,25,26,16,35
```

The immediate *non-abstract* successor of *AbstractMap* is HashMap, hence, it will inherit these paths. In the implication block of HashMap (*(this in java_util_HashMap - serl_map_FieldAddedMap) => (. . . or . . .)*), we only have two *fact blocks* seperated by *or*. The first *fact block* only has one *fact this = that* representing path 1. The second *fact block* has three *facts* (*!(this = that) and (that in java_util_Map) and (this.java_util_Map_map = that.java_util_Map_map)*) in which only the first two *facts* can be found in paths b, c, and d but not the last map comparison *fact*. This is where the pattern detector comes into play. The pattern detector is used by the code generator to detect the logic of map equality

discussed in Section 3.6.7 and abstracts the Java implementation to the last fact of map comparison. By doing this, we are basically uniting the different set of facts scattered over multiple paths into one *fact block*. We do similar abstraction for Arrays, Lists and Sets.

One-To-One Translation

This is a Java statement to an alloy statement (a *fact*) translation. There are four different kind of translations:

State Check Translation State check translation translates Jimple equality comparison between two fields or un-expanded method done by conditional operator (*== or !=*) or with comparision method (*equals*, *compareTo*) into alloy code. The State Equality Pattern detector (discussed in Section 3.6.4) is used for detecting such patterns. An example of such translation is *(this.serl_map_FieldAddedMap_field = that.serl_map_FieldAddedMap_field)* in a *fact block* of *implication* of *FieldAddedMap* class.

Type Check Translation Type check translation translates java type checking into a corresponding alloy *fact*. Type Checking Pattern detectors (discussed in Section 3.6.3 are used to detect such patterns. An example of such translation for *FieldAddedMap* is *(that in java_util_Map)*. We handle four different ways of type checking. Lets consider Figure 3.33 for a sample type hierarchy. Assuming super type B that overrides the *equals* method of *java.lang.Object*, and S and T as subtype of B and U as subtype of S all that inherit the *equals* method of B. The alloy translation for each cases of type checking in the *equals* method of B is shown as follows:

```
Object
  <- B [Defines m()]
       <- T
       <- S
            <- U [Overrides m()]
```

Figure 3.33: A type hierarchy for type check translation explanation.

- Java statement *that instanceof B* is translated as *(that in B)* in alloy.

- Java statement *this.getClass() == that.getClass()* is translated as *(that in B - S - T)* in alloy.
- Java statement *this.getClass() == S.class* is translated as *(that in S - U)* in alloy.
- Java statement *that = (B)parameter* is translated as *(that in B)* in alloy.

Dynamic Dispatch Translation Dynamic dispatch translation is essentially a type check translation that will limit the type of receiver object to a set of identified dispatch type for that method. For instance, let's assume a method *m()* in B is overridden by U as shown in Figure 3.33. Let's assume the type of receiver object is resolved to be B at call site, then a call to *that.m()* will result in generation of *(that in B-U)* fact enforcing that call to *m()* method for the receiver *that* can be any type $\{B, S, T\}$ but not $\{U\}$ since this method is overridden in U.

3.7.4 Type Exclusion Specification

This block is used to filter out all of the *interface* and *abstract* types from being considered for counterexample. The reason behind doing this is that the interface and abstract types cannot be initialized in real life. Furthermore, while processing pattern detection, there may be some code patterns that are not handled by the detector. We include such types in this block because we cannot represent its *equals* implementation logic. Nevertheless, we do report such problems.

3.7.5 Equivalence Property Specification

This block simply represents the three properties of equivalence relation: *reflexive*, *symmetric* and *transitive* properties using the defined equality predicate. The block serves as standard specification for the correctness of the *equals* method.

3.7.6 Validity Check

Validity checking (or Model Checking) is done in this block. To detect a *reflexive* violation we need *one* object, to detect a *symmetric* violation we need *two* objects and to detect a

transitive violation we need *three* objects. This check will return a counterexample for the case when the *equals* implementation is violated.

Chapter 4

Equals Checker Validation

4.1 Introduction

We have run the equals checker in several open source real world projects. The checker has successfully identified many equality checking patterns discussed in Chapter 2 (e.g. arrays, list, maps, sets, etc) and more importantly discovered several problems related to equivalence relation in the implementations. In this chapter, we will categorize problems reported by the checker on the basis of root causes, propose solution to these problems and elaborate on false positives and their causes. This will be followed by the explanation of code patterns that are not handled by the checker and their root causes.

4.2 Analysis Projects

Among different Java-based projects that have been analyzed, we found Tomcat 6 [5], Lucene 3.0 [4] and JDK 1.5 [42] as representative of most of the problems identified by the checker. Tomcat 6 is a web sever for hosting servlets, Lucene serves as search engine for various websites and JDK 1.5 is a Java development kit that provides important libraries for data structure collections, IO, network, security, graphics, etc. In Table 4.1, we have summarized the size of these projects in terms of classes, interfaces and number of equals

Description	Tomcat 6	Lucene 3.0	JDK 1.5
Interfaces	175	76	1745
Classes	1236	888	11239
Type hierarchies involving equals method	28	67	487
Classes overriding equals method	29	96	615
Total equals method expansion	31	98	698

Table 4.1: Summary of inspected projects.

implementations[1]. JDK 1.5 is the largest of all with *11,239* classes out of which *615* override the *equals* method. Altogether, *487* type hierarchies involve with these classes. Sometimes, a super *equals* method makes a call to methods that are overriden in subclasses. In such scenarios, the *equals* implementation for subclasses has to be re-examined. In JDK 1.5 altogether *698* classes were examined for a direct or inherited implementation of the *equals* method. Tomcat 6 is the smallest of all in terms of examined *equals* implementations(*31*) which is fewer than Lucene(*98*), however in terms of total classes and interfaces Tomcat is larger.

4.3 Path Generation and Filtering

The inter-procedure path generator produces several paths while expanding the *equals* method. Some of these paths are useful while others are useless and are pruned. Pruning is done applying different filters (implemented using flow analysis [1]) on the paths as they are are generated. These filters are as follows:

BooleanFilter Boolean filter is a boolean value *copy propagation analysis* run over generated paths of the *equals* method. A conflict in boolean value when detected in any intermediate path will result in pruning that path by this filter. A conflict in such a case could be a variable evaluated true at one point of a path but later evaluated

[1]Note that in Chapter 2, Lucene 2.4 was used that is replaced by Lucene 3.0, the newer version. Similarly, previous checkers could work in the presence of some compile errors but the current checker uses Soot [44] and requires these compile error to be fixed for proper analysis. While doing that some of the JDK 1.5 non public classes have been deleted due to excessive compile errors while porting to Eclipse [17] project environment.

as false without any assignment in between. It also eliminates *dead code* related to boolean variables and *false* returning paths.

NullnessAnalysisFilter NullnessAnalysisFilter is null constant copy propagation analysis. Like Boolean Filter a path having a conflict in *null* value is pruned.

TypeAnalysisFilter Type analysis filter prunes the paths containing conflicting types for a variable.

ThrowAnalysisFilter Throw analysis filtering prunes the paths that throw exception.

Redundant *Type.class* Filtering We have observed *Java Virtual Machine* injecting some dynamic type resolution code for static class field (*Type.class*) in type checking statements like *that.getClass() == Type.class*. Some of these paths are redundant and are pruned using this filter.

Description	**Tomcat 6**	**Lucene 3.0**	**JDK 1.5**
Total Paths	643	1232	49241
Boolean Path Filtering	473	763	30151
Nullness Analysis Filtering	0	0	845
Type Analysis Filtering	0	0	670
Throw Analysis Filtering	0	0	4440
Redundant Type.class Filtering	0	0	0
Other Intermediate Paths	61	136	7744
Final Working Paths	**109**	**333**	**5391**

Table 4.2: Summary of path generation and filtering.

The summary of path generation for the three project is shown in Table 4.2. The main problem with path based analysis has been considered to be path explosion. However, our approach of pruning paths whenever possible by applying these filters prune more than *80%* of the useless paths in all of these projects. Note that there could have been exponentially more paths than the total number of paths shown for each project. Since the filters are applied on each intermediate path after a change (method expansion), most of the useless paths are pruned in the earlier phase of path generation. Hence, even for a large project

like JDK, we have merely few thousands paths to be considered for pattern detection and Alloy [27] code generation.

4.4 Detectable Patterns

There are 7 different equality checking patterns that the checker is capable of detecting. These patterns have also been discussed in Section 3.6:

Type Checking Type checking pattern checks the type of parameter passed to the *equals* method. *e.g. (o instanceof Type), this.getClass() == that.getClass(), etc.*

State Comparison State comparison pattern checks fields and method comparison. *e.g. this.field == that.field, this.getField().equals(that.getField()), etc.*

State Comparison Involving Null Constant Sometimes state fields are checked for null value before performing a regular field comparison that might result in a path with the following code pattern: *if(this.field == null && that.field == null) return true;*. We translate such pattern as valid state check pattern i.e. *(this.field == that.field).*

Array Comparison This pattern contains logic of comparing two arrays.

List Comparison This pattern contains logic of comparing *java.util.List* types.

Set Comparison This pattern contains logic of comparing *java.util.Set* types.

Map Comparison This pattern contains logic of comparing *java.util.Map* types.

Table 4.3 summarizes the number of detected patterns in the project. Note that these patterns are accumulated per path basis. For clarity, a state comparison involving a field might appear in two paths of the same equals method. Even though the same field is being compared, we consider each path at a time and we will record two state comparison patterns even though we are comparing only one field in the class. So, the associated numbers represent behavior in the paths rather than the structure in the *equals* method.

Description	Tomcat 6	Lucene 3.0	JDK 1.5
Type Checking	254	570	4409
State Comparison	256	935	7029
State Comparison Involving Null Constant	83	165	831
Array Comparison	30	8	2513
List Comparison	0	0	117
Set Comparison	0	0	6
Map Comparison	6	0	108
Total Detected Patterns	**629**	**1678**	**15013**

Table 4.3: Summary of detected patterns.

4.5 Detected Problems

We have discussed the equivalence contract for the *equals* method in Section 1.1. There are three properties associated with an equivalence relation: *reflexive*, *symmetric* and *transitive* properties. The checker primarily reports on the possibility of violation of these properties in the implementation of the *equals* method. Besides these, the checker also reports on several other issues related to the *equals* implementation. A summary of all the problems reported by the checker is shown in Table 4.4 and a complete list of reported errors and warnings can be found in Appendices B and C, respectively.

4.5.1 Reflexive Property Violation

A reflexive property violation occurs when an object is not equal to itself. To illustrate on this kind of violation, lets consider the *java.net.InetAddress* class from JDK 1.5:

```
public class InetAddress implements ... {
  /**
   * Compares this object against the specified object. The result is true if and only
   * if the argument is not null and it represents the same IP address as this object. ...
   */
  public boolean equals(Object obj) {
    return false;
  }
}
```

The implementation of the *equals* method in this case always returns false. Hence an object of *InetAddress* will not be equal to any other object including itself. This clearly

Description	JDK 1.5		Lucene 3.0		Tomcat 6	
	Prob.	False Pos.	Prob.	False Pos.	Prob.	False Pos.
Reflexive Property Violation	3	2	0	0	0	0
Symmetric Property Violation	10	0	1	0	1	0
Transitive Property Violation	8	2	0	1	1	0
Unguarded null parameter	20	0	6	0	0	0
Probable ClassCastException	18	0	1	0	0	0
Similarity for Equality	13	0	0	0	0	0
State comparison on same object	3	0	0	0	0	0
Equals overloading for similarity	15	0	0	0	4	0
Equals overloading without overriding	22	0	0	0	4	0
This reference type test	0	1	0	0	0	0
Total Problems	**112**	**5**	**8**	**1**	**10**	**0**
Suspicious Implementation Warnings						
Use of non-equals method on field	5	0	0	0	0	0
Domain specific map implementation	5	0	0	0	0	0
Wrapper implementation pattern	15	0	0	0	3	0
Equals overloading with overriding	43	0	1	0	1	0
Path generation reaching cut-off	11	-	0	-	0	-
Total Suspicious Implementations	**79**	**0**	**1**	**0**	**4**	**0**

Table 4.4: Summary of detected problems.

violates the reflexive property of the equivalence relation. Furthermore, javadoc describes an implementation of the *equals* method which should have compared the states of *InetAddress* objects before returning true or false. We found 2 classes in JDK 1.5 that always return false and a case that always throws *UnsupportedOperationException* in *equals* method.

False Positives

The checker reports *two* false positives for the reflexive property violation in JDK 1.5. Both of these are due to an imprecise state space assumption in abstraction. We assume that if a state is operated on then a resulting new state may or may not be equal to the state before operation. For instance, the *this.field / 2 == that.field / 2* expression is checking if the result of the division operation on the fields is equal or not. We abstract such a comparison in alloy as *this.DIV_field_2 = that.DIV_field_2* which is basically a prefix representation of an expression on each side. The effect of such an abstraction is that it creates a new state different from previous state. Nevertheless, if the result of the operation has the same value

as the previous state then we have an imprecise assumption on the state space and may result in a false positive. For instance, let's consider the *com.sun.jmx.snmp.IPAcl.GroupImpl* class:

```
class GroupImpl extends PrincipalImpl implements ... {
  // Constructs a group using the specified subnet mask
  public GroupImpl (String mask) throws ... {
    super(mask);
  }

  public boolean equals (Object p) {
    if (p instanceof PrincipalImpl || p instanceof GroupImpl) {
      if ((super.hashCode() & p.hashCode()) == p.hashCode()) return true;
      else return false;
    }
    else { return false; }
  }
}
```

GoupImpl class represents a group of hosts in a subnet and is constructed using a subnet mask (e.g. *255.255.255.0 mask* represents a *256* available host in a traditional *Class C* network). The alloy model for *GroupImpl* is shown in Figure 4.1[2]. There are two things to consider. First, the *hashCode()* method is a special function and is represented as a state of the *java.lang.Object* class and translated as *Object_hashcode* field. The operation *super.hashCode() & p.hashCode()* is abstracted as a new state *Object_AND_hashcode_hashcode* which is a prefix representation of the binary expression. Our assumption for such an abstraction is that an operation on an existing state will result in a new state which may or may not be equal to the previous state. Hence, for the case when they are not equal, Alloy reports a reflexive property violation as shown in the counterexample of Figure 4.2. Nevertheless, for this.equals(this), *super.hashCode() & p.hashCode()* is exactly equal to *super.hashCode()* and hence we do not have a reflexive property violation in the class. This class, however, has a symmetric and a transtive property violation that will be discussed in the following sections.

[2]Note that we have changed the generated alloy model to make it fit in the available space.

```
abstract sig Object {
   Object_hashcode : Int,
   Object_AND_hashcode_hashcode : Int
}
sig PrincipalImpl extends Object { }

sig GroupImpl extends PrincipalImpl { }

pred Object :: equals( that: Object ) {
   ( this in GroupImpl ) =>
   (
      ( that in PrincipalImpl )
      and
      ( this.Object_AND_hashcode_hashcode = that.Object_hashcode )
   )
}

sig ExclusionSet in Object { }
fact { PrincipalImpl - GroupImpl in ExclusionSet }

assert reflexive { all a : Object - ExclusionSet | a.equals[a] }
assert symmetric { all a, b: Object - ExclusionSet | a.equals[b] <=> b.equals[a] }
assert transitive { all a, b, c: Object - ExclusionSet | a.equals[b] and b.equals[c] => a.equals[c] }

check reflexive for 1
check symmetric for 2
check transitive for 3
```

Figure 4.1: Alloy Model for GroupImpl class.

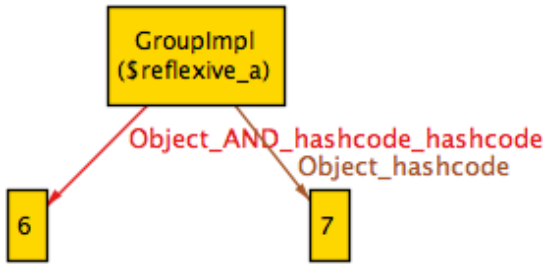


Figure 4.2: Counterexample for reflexive property violation in GroupImpl class.

4.5.2 Symmetric Property Violation

A symmetric property violation happens for two objects a and b when *a.equals(b)* is true and *b.equals(a)* is false or vice versa. There are several cases of symmetric property violation reported in the preliminary case study of Section 2.2 for JDK 1.5 that are also detected by the checker. Here, we present two instances of symmetric property violation one from JDK 1.5 and another from Tomacat 6. Lets consider an Alloy model for the *GroupImpl* class from JDK 1.5 shown in Figure 4.1. The symmetric property violation is shown in the counterexample of Figure 4.3. The counterexample is basically telling us that we will get a symmetric

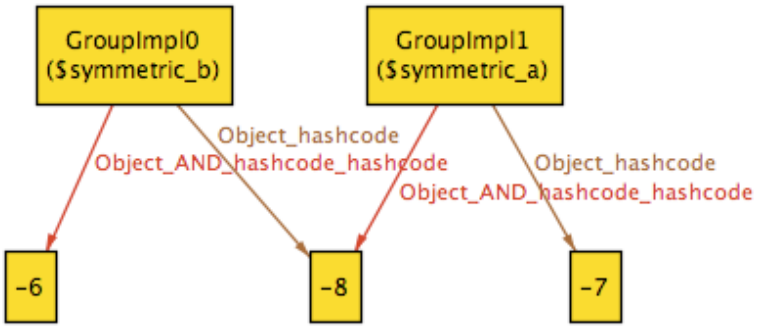


Figure 4.3: Counterexample for symmetric property violation in GroupImple class.

property violation when two objects of the *GroupImpl* class are compared with the *equals* method where both objects have the same value for the *Object_AND_hashcode_hashcode* field but different value for the *Object_hashcode*. Since the boolean expression with the bitwise-and operator in the *equals* has been abstracted to a new state, it is not directly clear how we get such a violation in real code. To illustrate the meaning with real code, let's look at the following code snippet:

```
GroupImpl group1 = new GroupImpl("255.255.255.128");
GroupImpl group2 = new GroupImpl("255.255.255.0");
group1.equals(group2); // Returns true
group2.equals(group1); // Returns false
```

Here, the subnet mask *255.255.255.128* for *group1* in a class C network represents *2* possible subnets with *128* available hosts each and the subnet *255.255.255.0* for *group2* represents *1* possible subnet with *256* available hosts. The *hashCode()* method for *GroupImpl* calls *super.hashCode()* that returns a *32* bit representation of the supplied mask. Hence, the *8* bit *LSB* (Least Significant Bits) for *group1*, say $LSB_8(group1)$, is *10000000* and for *group2*, say $LSB_8(group2)$, is *00000000*. Hence $LSB_8(group1); \& LSB_8(group2)$ is equal to $LSB_8(group2)$ but $LSB_8(group2) \& LSB_8(group1)$ is not equal to $LSB_8(group1)$ all *MSB*s (Most Significant Bit) being equal leads to the symmetric property violation. This also means that *group2* contains all of the hosts contained in *group1* but the opposite is not true. Hence, the lack of a bidirectional containment relation in the equivalence relation is causes the symmetric property violation in *GroupImpl* class. Here we have a fundamental

problem in the logic of the equivalence relation for *GroupImpl*. One possible fix for this problem would be to move this containment relation to another method and implement an equivalence relation on mask.

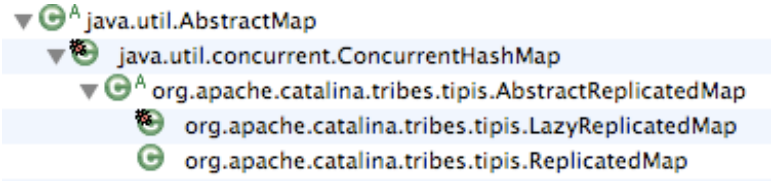


Figure 4.4: Type hierarchy associated with AbstractReplicatedMap class.

We will now present yet another case of symmetric violation from Tomcat 6. Let's consider the type hierarchy associated with the *org.apache.catalina.tribes.tipis.AbstractReplicatedMap* class from Tomcat 6 shown in Figure 4.4. *AbstractReplicatedMap* inherits from Java util's *ConcurrentHashMap*. *ConcurrentHasMap* inherits the *equals* implemenation from *AbstractMap*. Similarly, *LazyReplicatedMap* and *ReplicatedMap* inherit the *equals* method implementation from *AbstractReplicatedMap*. *AbstractReplicatedMap* overrides the *equals* implementation of *AbstractMap* as shown in Figure 4.5.

```
public abstract class AbstractReplicatedMap extends ConcurrentHashMap implements ... {
  protected transient byte[] mapContextName; // State is represented as array

  public boolean equals(Object o) {
    if (o == null) return false;
    if (!(o instanceof AbstractReplicatedMap)) return false;
    if (!(o.getClass().equals(this.getClass()))) return false;
    AbstractReplicatedMap other = (AbstractReplicatedMap) o;
    return Arrays.equals(mapContextName, other.mapContextName);
  }
  ... // Rest of code
}
```

Figure 4.5: Equals implementation of AbstractReplicatedMap.

The equality implementation of *AbstractMap* has already been discussed in Section 3.6.7 using Figure 3.29. The implementation of *AbstractReplicatedMap*, however, is not a regular map pattern and uses array comparisons. The *Map* hierarchy of the *java.util* package is a type compatible hierarchy which is inherited by *AbstractReplicatedMap* but introduces type-incompaitble sub-hierarchy by performing an *o.getClass().equals(this.getClass())* check. Also note that the *instanceof* check before the *getClass* check is redundant because

the latter is stronger type checking than the former. With the given implementation, a *ConcurrentHashMap* with the same state will be equal to a subtype of *AbstractReplicatedMap* but the opposite is not true:

```
ConcurrentHashMap hashMap = new ConcurrentHashMap();
LazyReplicatedMap replicatedMap = new LazyReplicatedMap(...);
hashMap.put("Key1", "Value1");
replicatedMap.put("Key1", "Value1");
hashMap.equals(replicatedMap); // returns true
replicatedMap.equals(hashMap); // returns false
```

Figure 4.6 shows Alloy model for the type hierarchy. We are not using a fully-qualified name for the sake of fitting the figure in the available space. In the implication block (defined in Section 3.7) of *ConcurrentHasMap* we see regular map fields being compared but for *ReplicatedMap* and *LazyReplicatedMap* we see the *mapContextName* array being compared. Hence, the state of *AbstractMap* and *AbstractReplicatedMap* are totally different objects. In such a scenario, ReplicatedMap should have composed ConcurrentHashMap rather than extending it.

Our checker has reported a symmetric property violation in *10* type hierarchies in JDK 1.5, *1* in Lucene and *1* in Tomcat 6.

4.5.3 Transitive Property Violation

A Transitive property violation happens for three objects *a*, *b* and *c* when *a.equals(b)* is true, *b.equals(c)* is true but *a.equals(c)* is not true.

We will now discuss an instance of probable transitive property violation in the class *com.sun.jndi.ldap.LdapName.TypeAndValue* shown in Figure 4.7. *LdapName* implements compound names for *LDAP v3* as specified by *RFC 2253*. The *TypeAndValue* inner class represents an attribute type and its corresponding value. Inside the *equals* method there are three kinds of state comparison: *type* field, *value* field and *getValueComparable()* method. An object of *TypeAndValue* type is equal to another object of the same type if either the *type* and *value* fields are equal or the *type* and return values of the *getValueComparable()* methods are equal and the *value* fields are unequal. Since a call to the *getValueComparable()*

```
abstract sig Object { }

sig Map in Object {
   map : Int -> Int
}

abstract sig AbstractMap extends Object { }
fact { AbstractMap in Map }

sig ConcurrentHashMap extends AbstractMap { }

abstract sig AbstractReplicatedMap extends ConcurrentHashMap {
   mapContextName : seq Int
}

sig LazyReplicatedMap extends AbstractReplicatedMap { }

sig ReplicatedMap extends AbstractReplicatedMap { }

pred Object :: equals( that: Object ) {
   ( this in ConcurrentHashMap - AbstractReplicatedMap - LazyReplicatedMap - ReplicatedMap ) =>
   (
      ( this = that )
      or
      (
         !( this = that )
         and
         ( that in Map )
         and
         ( this.map = that.map )
      )
   )
   else
   ( this in LazyReplicatedMap ) =>
   (
      ( that in LazyReplicatedMap )
      and
      ( this.mapContextName = that.mapContextName )
   )
   else
   ( this in ReplicatedMap ) =>
   (
      ( that in ReplicatedMap )
      and
      ( this.mapContextName = that.mapContextName )
   )
}
// Makes Interfaces and Abstract types to be not considered for counter example
sig ExclusionSet in Object { }
fact { AbstractMap - ConcurrentHashMap - AbstractReplicatedMap -
                                    LazyReplicatedMap - ReplicatedMap in ExclusionSet }
fact { Map - AbstractMap - ConcurrentHashMap - AbstractReplicatedMap -
                                    LazyReplicatedMap - ReplicatedMap in ExclusionSet }
fact { AbstractReplicatedMap - LazyReplicatedMap - ReplicatedMap in ExclusionSet }

assert reflexive { all a : Object - ExclusionSet | a.equals[a] }
assert symmetric { all a, b: Object - ExclusionSet | a.equals[b] <=> b.equals[a] }
assert transitive { all a, b, c: Object - ExclusionSet | a.equals[b] and b.equals[c] => a.equals[c] }

check reflexive for 1
check symmetric for 2
check transitive for 3
```

Figure 4.6: Alloy model for AbstractReplicatedMap type hierarchy.

```
  static class TypeAndValue {
    private final String value;
    private final boolean binary;
    private final boolean valueCaseSensitive;

    public final boolean equals(Object obj) {
      if (!(obj instanceof TypeAndValue)) { return false; }
      TypeAndValue that = (TypeAndValue)obj;
      return (type.equalsIgnoreCase(that.type) &&
        (value.equals(that.value)  || getValueComparable().equals(that.getValueComparable())));
    }
    ... // Rest of TypeAndValue code
  }
```

Figure 4.7: TypeAndValue Class.

```
abstract sig java_lang_Object { }
sig TypeAndValue extends java_lang_Object {
   value : Int,
   valueComparable : Int,
   type : Int
}
pred java_lang_Object :: equals( that: java_lang_Object ) {
   ( this in TypeAndValue ) =>
   ( (
         ( that in TypeAndValue )
         and
         ( this.type = that.type )
         and
         !( this.value = that.value )
         and
         ( this.valueComparable = that.valueComparable )
      )
      or
      (
         ( that in TypeAndValue )
         and
         ( this.type = that.type )
         and
         ( this.value = that.value )
      )
   )
}
... // Spec and check left out
```

Figure 4.8: Alloy model for TypeAndValue Class.

method for both the *this* and *that* receiver resolve to the same method object, this call is not expanded and treated as a new field *valueComparable*. A manual inspection of the code also revealed that *getValueComparable()* apart from depending on the *value* field also depends on other fields (*binary* and *valueCaseSensitive*). An Alloy model for this class is shown in Figure 4.8 and a counterexample is shown in Figure 4.9. The counter example shows for three objects of type *TypeAndValue(int type, int value, int valueComparable)*: *a(6, -8, -3)*, *b(6, -8, -5)* and *c(6, -4, -5)*, *a.equals(b)* is true and *b.equals(c)* is true but *a.equals(c)* is not

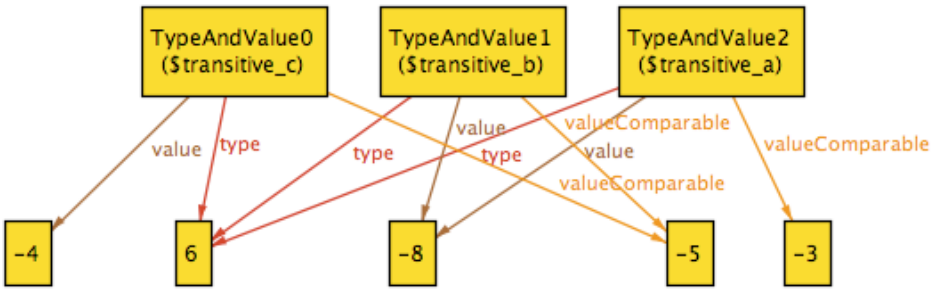


Figure 4.9: Counter example for transitive property.

```
final class NotifierArgs {
  int mask;
  String name;
  SearchControls controls;

  public boolean equals(Object obj) {
    if (obj instanceof NotifierArgs) {
      NotifierArgs target = (NotifierArgs)obj;
      return mask == target.mask && name.equals(target.name)
      && filter.equals(target.filter) && checkControls(target.controls);
    }
    return false;
  }

  private boolean checkControls(SearchControls ctls) {
    if ((controls == null || ctls == null)) {
      return ctls == controls;
    }
    return (controls.getSearchScope() == ctls.getSearchScope()) &&
      ... // Other state comparison between controls and ctls;
  }
}
```

Figure 4.10: Equals implementation of NotifierArgs Class.

true.

The checker has reported a transitive property violation in *10* type hierarchies in JDK 1.5, *1* in Tomcat 6 and *1* in Lucene. Upon a manual inspection of these reported problems, we found *2* false positive in JDK 1.5 and *1* in Lucene.

False Positives

Sometimes the abstraction of code does not match its real behavior due to the limitation of pattern detection algorithms. In such a scenario chances of getting a false alarm also increases. For instance, let's consider the *com.sun.jndi.ldap.NotifierArgs* class from JDK 1.5 shown in Figure 4.10.

The *NotifierArgs* class holds necessary information for an event registration or de-registration request in an *LDAP* server. This class composes the *javax.naming.directory .SearchControls* class which is responsible for determining scope and return value of a search. *SearchControls*, however, does not have an *equals* method implemented on it. The *equals* method of *NotifierArgs* performs regular state checking on the *mask*, *name* and *filter* fields. When it comes to comparing *controls*, *NotiferArgs* has to define its own method for the task. This is done in *checkControls* method where first true-returning path check if both the controls are *null*. The second true-returning path checks if *this.controls* is *null* and the third path checks if *that.controls* is *null* followed by *searchScope* and other state comparison in both the second and third path. Nevertheless, the checker cannot classify the *null* checking pattern as a valid *null* checking pattern because the logic is in the combination of both *if* and *return* statements. In such a scenario, *null* checking is ignored and Alloy generates *this.controls == that.controls* for the first path. Furthermore, in Alloy model we represent a state comparison as a field comparison of composing class and not of composed class. This is because all fields in Alloy are represented as integer and require abstraction of multi-level composition into a single level. Hence, the translation of *this.controls.getSearchScope()* in Alloy becomes *this.controls_searchScope*. Assuming that *mask*, *name* and *filter* are compared in all three paths, we present a modified version of the generated Alloy model to show how the transitive property gets violated as shown in Figure 4.11[3].

```
abstract sig Object { }
sig NotifierArgs extends Object {
   controls : Int,
   controls_searchScope : Int,
}
pred Object :: equals( that: Object ) {
  ( this in NotifierArgs ) =>
  (( that in NotifierArgs ) and ( this.controls = that.controls ))
  or
  (( that in NotifierArgs ) and ( this.controls_searchScope = that.controls_searchScope ))
  or
  (( that in NotifierArgs ) and ( this.controls_searchScope = that.controls_searchScope ))
}
... // Spec and Check bloc left out
```

Figure 4.11: Modified alloy model for NotifierArgs Class.

[3]Note that we are ignoring *mask*, *name* and *filter* comparison assuming they are compared in all paths.

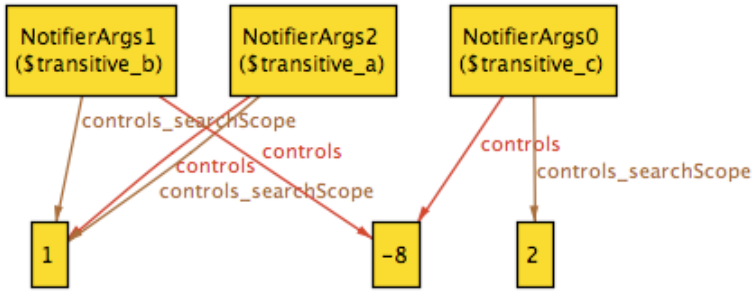


Figure 4.12: Counter example for transitive property.

Figure 4.12 shows the counterexample generated by Alloy. However, in real life this never happens because the logic is sound where either both *this.controls* and *that.controls* are *null* or both are *not null* and have equal states. This abstracts to *this.controls* and *that.controls* being equal in all paths in real life. Hence, the false positive here is because of imprecise abstraction of actual code behavior.

4.5.4 Unguarded null parameter

The equals contract specifies that an *equals* method must return false if the parameter passed to the method is *null*. We found several cases in the *equals* method where this condition is not checked. For instance, let's consider the *equals* method of *org.apache.lucene.search .function.ValueSourceQuery* from Lucene 3.0.

```
public boolean equals(Object o) {
  if (getClass() != o.getClass()) { return false; }
  ValueSourceQuery other = (ValueSourceQuery)o;
  ... // Rest of the code
}
```

The call to the *equals* method will result in a *NullPointerException* if *o* is *null* when the *o.getClass()* method in the first line is invoked. We found *20* such cases in JDK 1.5 and *6* cases in Lucene 3.0. The solution to this problem is to guard the parameter with a null check (ie. *if(o == null) return false;*) before dereferencing parameter *o*.

4.5.5 Probable ClassCastException

The equals method usually contains a type checking guard before casting the *Object* type parameter to another type. If such a guard is not present or an improper guard is present then a *ClassCastException* is thrown if the parameter is of incompatible type to the cast type. For instance, let's consider the *org.apache.lucene.analysis.tokenattributes.PayloadAttributeImpl* class in Lucene 3.0:

```
public class PayloadAttributeImpl extends AttributeImpl implements PayloadAttribute ... {
  public boolean equals(Object other) {
    if (other == this) { return true; }
    if (other instanceof PayloadAttribute) {
      PayloadAttributeImpl o = (PayloadAttributeImpl) other;
      ... // Rest of the code
    } ...
  }
}
```

The *PayloadAttributeImpl* class implements the *PayloadAttribute* interface which is also used in *instanceof* type checking. However, the *PayloadAttribute* interface is also implemented by the *org.apache.lucene.analysis.Token* class. If an object of *PalyloadAttributeImpl* is compared to an object of *Token* class, a *ClassCastException* will be thrown. We found *18* cases of similar problems in JDK 1.5 and *2* cases in Lucene 3.0.

4.5.6 Similarity implementation for equality

Equals methods in most cases are designed to compare objects within the same type hierarchy. Nevertheless, this does not prevent programmers from going beyond the type hierarchy for equality comparison. Furthermore, all of the non-primitive types are subtypes of the *java.lang.Object* class in Java and can be considered as falling within the type hierarchy of the *Object* class. In that respect, special care must be taken for *adapter pattern* [22] where the *equals* method of adapter delegates equality comparison to the *equals* method of the adaptee which can potentially cause a symmetric property violation. For instance, lets consider the *equals* method of the *com.sun.corba.se.impl.orbutil.RepIdDelegator* class from

JDK 1.5:

```
public final class RepIdDelegator implements ... {
  // RepositoryId does not belong to type hierarchy of RepIdDelegator
  private RepositoryId delegate;
  public boolean equals(Object obj) {
    if (delegate != null) return delegate.equals(obj);
    else return super.equals(obj);
  }
}
```

The *RedIdDelegator* class has a *delegate* field of type *com.sun.corba.se.impl.util.RepositoryId* and the comparison *delegate.equals(obj)* compares an object of *RepositoryId* with *RedIdDelegator*. However, *RepositoryId* does not implement the *equals* method. Hence, an object of *RepIdDelegator* may be equal to an object of *RepositoryId* but the opposite is not true. This may lead to a symmetric property violation. Liskov et. al. classify such an equality comparison as *similarity* comparison and proposes to implement a separate similarity method [30]. A proper implementation of a similarity pattern in JDK 1.5 can be found in the *java.lang.String* class where instead of implementing similarity logic in the *equals* method, the *String* class defines the *boolean contentEquals(java.lang.StringBuffer)* method to compare similarity between *String* and *StringBuffer*.

```
private static final class XDouble {
  private double value;

  public boolean equals(Object val) {
    ...
    XDouble oval = (XDouble)val;
    ...
    if (value != value && oval.value != oval.value) return true;
    return false;
  }
}
```

Figure 4.13: Equals implementation of the XDouble class.

4.5.7 State comparison on same object

In general, state comparison is done between the *this* and *that* object. We found some redundant state comparisons where *this.field* is compared to *this.field* and/or *that.field* is compared to *that.field*. For instance, let's consider the *com.sun.org.apache.xerces.internal.impl*

.dv.xs.DoubleDV.XDouble class from JDK 1.5 shown in Figure 4.13.

In the code snippet, *this.value* is compared to *this.value* and *oval.value* is compared to *oval.value* which we believe should have been *this.value* compared to *oval.value*. We found *3* cases of similar comparison in JDK 1.5.

4.5.8 Equals overloading for similarity

We have found several instances of the *adapter pattern* [22] in JDK 1.5 that adapts the *equals* method of the adaptee in the adapter and overloads *equals* for checking similarity between adapter and adaptee. This may cause a symmetric property violation. For instance, let's consider the *sun.security.x509.AlgorithmId* (Adapter) and *sun.security.util.ObjectIdentifier* (Adaptee) classes. The *AlgorithmId* class overloads its *equals* method to be comparable to both *AlgorithmId* objects and *ObjectIdentifier* objects. *ObjectIndentifier*, on the other hand, has no idea about the *AlgorithmId* class:

```
// AlgorithmId class
public class AlgorithmId implements ... {
  public AlgorithmId(ObjectIdentifier oid) { // oid to be wrapped }
  public boolean equals(Object other) { ... // For Object }
  public boolean equals(AlgorithmId other) { ... // For AlgorithmId }
  public final boolean equals(ObjectIdentifier id) { ... // For ObjectIdentifier }
}

final public class ObjectIdentifier implements ... {
  public ObjectIdentifier (String oid) throws IOException { ... // Constructor }
  public boolean equals(Object other) { ... // For Object }
  public final boolean equals(ObjectIdentifier id) { ... // For ObjectIdentifier }
}
```

In such situation, a comparison between object of *AlgorithmId* and *ObjetIdentifier* may introduce symmetric property violation as follows:

```
ObjectIdentifier oid = new ObjectIdentifier("1.23.34.45.56");
AlgorithmId aid = new AlgorithmId(oid);
aid.equals(oid); // Returns true
oid.equals(aid); // Returns false
```

Note that this is not because of the problem in the implementation of the *boolean equals(Object)* method but because this *equals* method is shadowed by an overloaded version. We found *15* cases of such problems in JDK 1.5 mostly in *sun.security*, *sun.tools.tree* and *com.sun.org.apache.xpath* packages. In Tomcat 6, we found 4 cases of this problem mostly in the *org.apache.jasper.xmlparser* and *org.apache.tomcat.util.buf* packages. The solution is to change the name of the overloaded *equals* method to something else and call these methods when similarity check is needed instead of equality.

4.5.9 Equals overloading without overriding

A class of type T may choose not to override the *boolean equals(Object)* method but implement its own overloaded version: *boolean equals(T)*. In such scenario, a simple casting of argument at the call site from type T to an *Object* type will switch between the overloaded equality of T to the identity equality of *Object* that may result in an inconsistent answer. This may be unsafe if the programmer does not know the details of equality implementation in the class T. For example, let's consider *java.awt.geom.Area* class from JDK 1.5. The *Area* class creates an area geometry from the specifed *Shape* object. The overloaded *equals* for *Area* is as follows:

```
public class Area implements Shape, Cloneable {
  public Area(Shape s) { ... // Takes a shape as parameter to create area geometry }
  public boolean equals( Area other) { ... // Logic for comparing Area }
}
```

Since the parameter of the *equals* method is of type *Area*, the *equals* method of Object class is not overriden but overloaded. So, for the *Object* type parameter *Object.equals(other)* will be called and for the parameter of type *Area*, *Area.equals(other)* will be called. This results in following inconsistency:

```
Rectangle rect = new Rectangle(1,1,1,1);
Area area = new Area(rect);
Object object = new Area(rect);
area.equals(object); // returns false because of identity equality of Object class
area.equals((Area)object); // returns true because of value equality of Area class
```

A solution to this problem is overriding *equals* method in Area class with *boolean equals(Object)* signature and returning *false* if parameter is not of type *Area* instead of just overloading. Bloch et al. also discourage the use of equals overloading in [10, 11].

We found *22* classes in JDK 1.5 with this problem affecting mostly the *sun.tools.tree*, *com.sun.org.apache.xpath*, *sun.security.acl* and *com.sun.org.apache.xerces* packages. In Tomcat 6, we found *3* cases of this problem mostly in the *org.apache.tomcat.util.buf* package.

4.5.10 Type checking on this object

Conventionally, type checking is done on the object supplied in the parameter of the *equals* method. Type checking on *this* object is most probably a typographic error. We found *1* case in JDK 1.5 that type checks the *this* object instead of *that*. Upon manual inspection it turns out to be intentionally done. Nevertheless, as a coincidence, there is another problem associated with the code. Let's consider the *com.sun.org.apache.xml.internal.utils.synthetic .reflection.EntryPoint* class where all of these happen:

```
abstract public class EntryPoint implements Member {
  public boolean equals(Object obj) {
    EntryPoint otherep = null;
    if (obj instanceof EntryPoint)
      otherep = (EntryPoint) obj;
    else if (obj instanceof java.lang.reflect.Constructor || obj instanceof java.lang.reflect.Method)
        otherep = (EntryPoint) obj;
    return (otherep != null &&
      ((this instanceof Constructor && otherep instanceof Constructor) ||
      (this instanceof Method && otherep instanceof Method &&
      this.getName().equals(otherep.getName()))) &&
      otherep.getDeclaringClass().equals(declaringclass) &&
      otherep.getParameterTypes().equals(parametertypes));
  }
}
```

EntryPoint is an abstract class which has two concrete subclasses, *Constructor* and *Method*, in the same package i.e. *com.sun...reflection*. Before reaching the *this instanceof Constructor* check, the type check of *obj* in *obj instanceof java.lang.reflect.Constructor* followed by type casting to *EntryPoint* itself may cause a *ClassCastException* because

the programmer in this case is confusing the *Constructor* and *Method* classes from the *com.sun...reflection* package with the same named classes from the *java.lang.reflect* package that do not belong to the same type hierarchy. Secondly, the intention of the programmer is type incompatible equality i.e. both *this* and *that* must be of type *Constructor* or both be of type *Method* to be considered *equal* thus preventing inter-class equality. The explicit type checking for subtypes couples *EntryPoint* with all of its subtypes which could have been avoided simply by a *this.getClass() == that.getClass()* check. Even though we found only one case of such a problem, it is clearly an unorthodox type checking pattern and a potential sign of problem.

4.5.11 Use of non-equal methods on fields

Equals checker is a hierarchy based path analysis tool. We do not expand a method in a path that does not belong to any of the classes in the type hierarchy of the analysis class. This also means that we do not expand a method whose receiver object is a *member field*, or an *array element* of a member field or a *return value* of a member function which we collectively call as states of objects being compared. If we find a comparison between fields that is done not using an *equals* method or operators other than *== or !=*, we report such a comparison as a *warning*. Here, we are relying on the assumption that the *equals* method of non hierarchy objects are implemented correctly. We found a few cases in JDK 1.5 where neither *equals* method or *==* or *!=* operator were used for comparing fields. Lets consider *sun.security.x509.GeneralName* as an example:

```
public boolean equals(Object other) {
  if (this == other) { return true; }
  if (!(other instanceof GeneralName)) return false;
  GeneralNameInterface otherGNI = ((GeneralName)other).name;
  try {
    return name.constrains(otherGNI) == GeneralNameInterface.NAME_MATCH;
  } catch (UnsupportedOperationException ioe) { return false; }
}
```

The comparison *name.constrains(otherGNI)==GeneralNameInterface.NAME_MATCH* is too specific for the checker to generalize as a valid state comparison. We were expecting *name.equals(otherGNI)* comparison in this case. Hence, we warn users to manually inspect the code for problems. We found *5* cases of such comparisons in JDK 1.5.

4.5.12 Domain specific map implementation

The pattern detector for map works for a general map equality pattern where the *size* and *key value mapping* of the *this* and *that entry set* of both maps are checked for equality. If a map is implemented not using methods specified in *Map interface* and mapping not compared through iterators of *entry set* and none of the pattern detectors can identify the comparison logic then such implementations are flagged as potential problems. For instance, let's look at the *equals* method of *javax.management.openmbean.TabularDataSupport* from JDK 1.5:

```
public class TabularDataSupport implements TabularData, Map ... {
  public boolean equals(Object obj) {
    ...
    TabularData other;
    ...
    if ( ! this.getTabularType().equals(other.getTabularType()) ) { return false; }
    if (this.size() != other.size()) { return false; }
    for (Iterator iter = this.values().iterator(); iter.hasNext();  ) {
      CompositeData value = (CompositeData) iter.next();
      if ( ! other.containsValue(value) ) { return false; }
    }
    return true;
  }
}
```

TabularDataSupport implements a *Map* interface to provide tabular data support for the *sun.management.MappedMXBeanType* class. Its equality is defined in terms of *tabular type*, *size* of map and *containment of values*. The equality logic, however, lacks a *key to value mapping* check which is a necessary condition for a regular map comparison pattern. The map pattern detector cannot classify the statements in and within the loop to any known

pattern as it is a domain specific implementation. Such patterns are reported for manual inspection. We have altogether *5* similar cases including *java.util.IdentityHashMap*[4] from JDK 1.5.

Note that we also try to isolate domain specific List and Set patterns but could not find any in the analyzed projects.

4.5.13 Wrapper implementation pattern

The similarity pattern has an equality comparison with a field whose type does not belong to any type in the type hierarchy of the class being analyzed. The wrapper pattern is similar to the similarity pattern, the only difference being that the type of the field belongs to the type hierarchy of the composing class. For instance, let's consider the *java.util.Collections.SynchronizedList* class from JDK 1.5:

```
static class SynchronizedList<E> extends SynchronizedCollection<E> implements List<E> {
  // Adaptee or wrapped object
  List<E> list;
  SynchronizedList(List<E> list) {
    super(list);
    this.list = list;
  }

  public boolean equals(Object o) {
    synchronized(mutex) {return list.equals(o);}
  }
}
```

Here, the *SynchronizedList* class is providing synchronization to the regular *List* object passed in the constructor. Essentially, *SynchronizedList* is adapting the un-synchronized list for synchronized access. Similarly, the *equals* method is also adapted to delegate the equality comparison to the original list by calling *list.equals(o)*. The correctness of the Wrapper pattern depends on the correctness of the *equals* method of the wrapped field. In fact, we have found a symmetric property violation in the *List* type hierarchy in JDK

[4] *IndentityHashMap* is known to have symmetric property violation with other regular maps also shown in Section 2.2.1.

1.5 that may affect the equivalence relation between *SychronizedList* and other *List*s in the type hierarchy. We have found *15* cases of the wrapper implementation pattern in JDK 1.5. Note that we identify the wrapper pattern to inform users for probable problems and for manual inspection because we do not expand methods on fields and it is hard to reason about the correctness of an *equals* of a field object without proper expansion.

4.5.14 Overloaded equals method with overriding

The *equals* method has a very special purpose of comparing equality between two objects. Overloading the *equals* method for multiple purposes often leads to violation of the equals contract. The checker has reported *71* cases of *equals* overloading among which two of the categories have been discussed in Sections 4.5.8 and 4.5.9 that are potentially problematic. The less problematic case is the one when both overloading and overriding of the *equals* method is done. Upon manual inspection of these cases, we coincidentally found symmetric property violations in some of the classes. Consider *com.sun.java_cup.internal.lr_item_core* and *com.sun.java_cup.internal.lalr_item* in the *java_cup* package:

```
public class lr_item_core {
  public lr_item_core(production prod) { ... // Constructs LR item from production }
  public boolean equals(Object other) {
    if (!(other instanceof lr_item_core)) return false;
    else return equals((lr_item_core)other);
  }
  // Overloaded equals
  public boolean equals(lr_item_core other) { return core_equals(other); }
}
public class lalr_item extends lr_item_core {
  public boolean equals(Object other) {
    if (!(other instanceof lalr_item)) return false;
    else  return equals((lalr_item)other);
  }
}
public boolean equals(lalr_item other) { // Overloaded equals method
  if (other == null) return false;
  return super.equals(other);
}
```

For same *production* an *lr_item_core* object may be equal to *lalr_item* but the opposite may not be true because of the type checking in *lalr_item.equals(Object)*:

```
production p = ... // A production
lr_item_core lr = new lr_item_core(p);
lalr_item lalr = new lalr_item(p);
lr.equals(lalr); // Returns true
lalr.equals(lr); // Returns false
```

We found *2* type hierarchies in the *java_cup* packages totalling 6 classes that are associated with this violation. There are totally *43* cases of overloading with overriding in JDK 1.5

4.5.15 Path generation reaching cut-off limit

Path explosion is one major hurdle for a path based analysis. The *equals* method with a big body and multiple control flow branches can create path explosion. After experimenting with several limiting numbers for the total number of paths per *equals* method, we have decided upon a maximum path size limit of *500*. If a path generation reaches *500* paths for an *equals* method, we will stop further expansion and report such a case for manual inspection. This limit does not affect detectable patterns for the checker for all the three projects. An example of a case where expansion reached the cutoff limit is the equals method of the *com.sun.org.apache.xerces.internal.jaxp.datatype.XMLGregorianCalendarImpl* class. But because of its big size we are not presenting it here. We have altogether 11 cases from JDK 1.5 that reached this cutoff limit. Upon manual inspection of each case we figured they have complicated comparison logic that will anyway fall in the *Unknown Code Patterns* section that is discussed in Section 4.6. Nevertheless, the ability to expand *687* out of *698* equals methods (98.42 %) in JDK 1.5 and all of the *equals* methods in Lucene 3.0 and Tomcat 6 shows that path-based analysis can be a useful machinery for program analysis.

4.6 Unknown Code Patterns

We started off by thinking of the *equals* method as a simple method containing a type checking block followed by a state checking block where two fields or methods are compared using a standard relational operator like == or using standard methods like *equals* or *compareTo*. This assumption worked well enough for the projects like Lucene, Tomcat 6 and most of JDK 1.5. However, when it came down to JDK 1.5 whose source code is contributed by many programmers of varying skills and coding styles, we did find some interesting *equals* implementations that surprised our checker. In this section, we will present several categories of such implementations and classify them as *supportable* after some plumbing or *not supportable* due to fundamental problem in the logic of the checker's implementation. Table 4.5 summarizes our manual analysis of *unknown code patterns*. A complete list of unknown code patterns can be found in Appendices D and E.

Description	**JDK 1.5**	**Lucene 3.0**	**Tomcat 6**
Supportable Patterns			
Use of a field to represent an array length	13	2	0
Use of an array as a set	3	0	0
Multi-Dimensional array equality pattern	1	0	0
Collection operations on field	13	2	0
Handling data ow of boolean type	7	0	0
Control dependency	3	0	0
Total supportable patterns	**40**	**4**	**0**
Un-supportable Patterns			
Domain specific representation on array	6	1	0
Domain specific representation on field	9	1	0
Comparison delegated to a field	9	0	0
Polymorphic field	2	0	0
Creation of new state for equality	11	0	0
Wrapped state Comparison	2	0	0
Domain specific equality	1	0	0
Total un-supportable patterns	**40**	**2**	**0**
Total supportable and un-supportable	**80**	**6**	**0**

Table 4.5: Summary of unknown code patterns.

4.6.1 Use of a field to represent an array's length

In the array pattern, we expect *array.length* to be used for checking the array's bounds. In the absence of such a bound check we do not consider such a code pattern as a valid array pattern. For instance, let's consider *java.net.Inet6Address* class from JDK 1.5:

```
public final class Inet6Address extends InetAddress {
  final static int INADDRSZ = 16;
  byte[] ipaddress; // 16 bytes IPv6 address
  public boolean equals(Object obj) {
    if (obj == null || !(obj instanceof Inet6Address)) return false;
    Inet6Address inetAddr = (Inet6Address)obj;
    for (int i = 0; i < INADDRSZ; i++) {
      if (ipaddress[i] != inetAddr.ipaddress[i]) return false;
    }
    return true;
  }
}
```

The *for* loop in the code snippet compares a sentinel *i* to a constant *INADDRSZ* instead of *ipaddress.length*. Even though the code has a sequential array element comparison required for a valid array pattern, it is lacking *size check* and *bound check* to be classified as a valid array pattern. It is justified to use this constant in this case because the programmer of the code knows that IPv6 requires *16* bytes and the *ipaddress* array is exactly that size all the time for any object of *Inet6Address*. Nevertheless, the checker does not know of such an implicit assumption. It assumes that if *ipaddress.length* is not used for bound check then an *IndexOutOfBoundException* might be thrown thus breaking the *equals* method implementation and hence reports as *unknown code pattern*. We found *13* cases in JDK 1.5 and *2* cases in Lucene that use another field to represent the length of an array field. If we add comparison of sentinel with a member field as a valid bound check pattern and in its presence relax the size check constraint then we will be able to support all *13* cases as a valid array pattern by the checker.

4.6.2 Use of an array as a set

We have implemented the array equality pattern to detect sequential element comparison and implemented the set equality pattern to detect bidirectional containment relation for subtypes of *java.util.Set*. Nevertheless, we found cases in JDK 1.5 that use array for set representation. Let's consider a code segment from the *java.security.CodeSource* class as follows:

```
private boolean matchCerts(CodeSource that, boolean strict) {
  ...
  boolean match;
  ...
  {
    for (int i = 0; i < certs.length; i++) {
      match = false;
      for (int j = 0; j < that.certs.length; j++) {
        if (certs[i].equals(that.certs[j])) { match = true; break; }
      }
      if (!match) return false;
    }
    return true;
  }
  ...
}
```

The code is taken from the *matchCerts(CodeSource that)* method that is called from *equals* and returns true if all of the certificates(an array of *java.security.cert.Certificate* objects) in *this.certs* are also present in the *that.certs* object. It is possible to extend the array pattern detector to detect this pattern and classify it as a set pattern. The conditions we will be looking for are bound check on *this.certs*, bound check in *that.certs*, element comparisons that involve sentinels from both *this.certs.length* and *that.certs.length* bound checks and finally *this.certs* and *that.certs* element comparison. We found *3* cases of such patterns in JDK 1.5.

4.6.3 Multi-Dimensional array equality pattern

The arrays equality pattern considers only one dimensional sequences for comparison. We have found *one* case in JDK 1.5 that checks for equality of a *2* dimensional arrays or *matrix*. It is possible to extend one dimensional array comparison to a matrix equality pattern while enforcing similar constraints of one dimension array equality to both dimensions. The code snippet for matrix comparison of the *com.sun.corba.se.impl.ior.StubIORImpl* class is as follows:

```
public class StubIORImpl {
  private byte[][] profileData;
  public boolean equals(java.lang.Object obj) {
    ...
    StubIORImpl other = (StubIORImpl)obj;
    ...
    return ... && equalArrays( profileData, other.profileData );
  }

  // For each row check column equals by calling equalArrays([],[])
  private boolean equalArrays( byte[][] data1, byte[][] data2 ) {
    if (data1.length != data2.length) return false ;
    for (int ctr=0; ctr<data1.length; ctr++) {
      if (!equalArrays( data1[ctr], data2[ctr] )) return false ;
    }
    return true ;
  }

  // Regular array comparison
  private boolean equalArrays( byte[] data1, byte[] data2 ) {
    if (data1.length != data2.length) return false ;
    for (int ctr=0; ctr<data1.length; ctr++) {
      if (data1[ctr] != data2[ctr]) return false ;
    }
    return true ;
  }
}
```

The equals method in the code calls the *equalsArray([][],[][])* method that compares two matrices. The comparison of columns for both matrices inside *equalsArray([][],[][])* is further delegated to the *equalsArray([],[])* method which is a regular array equality pattern.

4.6.4 Collection operations on field

We hoped that the *equals* method while comparing a collection type field (*List*, *Map*, *Set*) would do so by using its well-defined *equals* method. Nevertheless, some of the classes in JDK 1.5 implemented *equals* by *re-inventing the wheel*. Let us look at the *equals* implementation of the *javax.naming.ldap.LdapName* class from JDK 1.5:

```
public class LdapName implements Name {
  private transient ArrayList rdns;
  public boolean equals(Object obj) {
    ...
    LdapName that = (LdapName)obj;
    ...
    if (rdns.size() != that.rdns.size()) { return false; }
    ...
    // Compare RDNs one by one for equality
    for (int i = 0; i < rdns.size(); i++) {
      Rdn rdn1 = (Rdn) rdns.get(i);
      Rdn rdn2 = (Rdn) that.rdns.get(i);
      if (!rdn1.equals(rdn2)) {  return false; }
    }
    return true;
  }
}
```

The field *rdns* is an *ArrayList* and instead of simply comparing *this.rdns.equals(that.rdns)*, the *equals* method of *LdapName* re-implements the logic of the equivalence relation for *List*. The checker expects such logic only in subtypes of *java.util.List* and for either the *this* or *that* object but not for fields. Nevertheless, it will be able to handle this by first relaxing the constraint that the list pattern can not only happen on *this* or *that* receiver objects but also on their fields. Second, the implementation is closer to the array equality pattern where array reference is replaced by an *rdns.get(i)* call. Hence, updating the list pattern to also accommodate the array pattern with the *get(int)* call will do the job. We found a similar implementation for *Map*s and *Enumeration* type fields. All of these can be handled either by improving the existing pattern detector for *list*, *map* and *set* patterns to work also on fields or in the case of *Enumeration* write a new pattern detector and represent the *Enu-*

meration pattern by abstracting as a field of the *set* type in the Alloy model. Altogether, *13* cases of such implementations were found in JDK 1.5 and *2* cases were found in Lucene 3.0.

4.6.5 Handling data flow of boolean type

The checker performs in-place evaluation of conditional expressions like == or *!*= in a statement whenever possible. For example, let us consider a null checking pattern that follows:

```
if(this.field == null && that.field == null) return true;
```

The conjunction of two conditional expression in the *if* statement is equivalently translated into two *if* statements in Jimple, each containing one conditional expression. So, an in-place evaluation of the conditional expression in each *if* statement can detect that *this.field* and *that.field* are being compared to a *null* constant. But this does not work all the time. Sometimes the programmer uses clever programing tricks that require further processing. For instance, in the *java.beans.PropertyDescriptor* class, the *equals* method delegates a *null* checking pattern for two state objects of type *Method* to the *compareMethod* method:

```
boolean compareMethods(Method a, Method b) {
  if ((a == null) != (b == null)) { return false; }
  if (a != null && b != null) {
    if (!a.equals(b)) { return false; }
  }
  return true;
}
```

In the code snippet, after the *if((a == null) != (b == null))* statement falls through, either both *a* and *b* are *null* or both are *not null* which is a valid *null checking pattern*. For a true returning path, the null checking expression must evaluate to false and the control must fall through. The checker needs to evaluate the *two* conditional expression operands followed by the *!*= operator in conjunction with the branching behavior of the

if statement to successfully detect such a pattern as a *null checking pattern*. Similarly, another interesting and repeating null checking pattern found in JDK 1.5 is the use of the *xor* operator. For instance, the *equals* method of the *java.rmi.dgc.VMID* class uses the *xor* operation to perform null checking:

```
public boolean equals(Object obj) {
  if (obj instanceof VMID) {
    VMID vmid = (VMID) obj;
    ...
    if ((addr == null) ^ (vmid.addr == null)) return false;
    if (addr != null) { ... // State comparison logic}
    return true;
  } else { return false; }
}
```

In the code snippet, after the *xor* statement is executed and control does not branch to the *return false;* statement, the *addr* and *vmid.addr* fields are either both *null* or both *not null*. Besides evaluation of the conditional expression associated with == and *!=*, we also need to evaluate boolean expressions associated with *xor*. So far, an in-place processing of the relational expression was enough to handle expressions like *this.field == that.field*, *that.field == null*, etc. But now, because of such null checking patterns we need an expression evaluation analysis on a data flow of boolean type to successfully handle such patterns. Altogether, 7 cases of similar *null checking patterns* were found in JDK 1.5.

4.6.6 Control dependency

It is generally useful to keep debug code snippets along with the real code. When this is done, a global *flag* is used to either enable or disable the debug code. For instance, the *equals* method of *javax.security.auth.PrivateCredentialPermission* class through intermediate method calls reaches the *impliesCredentialClass(String, String)* method, that has debug code with a *testing* field flag as switch to enable or disable printing of debug information (Figure 4.14).

Here, the check for testing the field does not contribute in any way to the equivalence relation and can be ignored. Hence, we need a control dependency analysis that can isolate

```
private boolean impliesCredentialClass(String thisC, String thatC) {
  ...
  if (testing)
    System.out.println("credential class comparison: " + thisC + "/" + thatC);
  ...
}
```

Figure 4.14: Implementation of the impliesCredentialClass() method.

the comparisons that contribute to the return value and ignore all those that do not. We found *3* cases of similar patterns in JDK 1.5.

4.6.7 Domain specific representation on array

The array equality pattern checks if two arrays are *equal* as a whole ie. the size of two arrays must be equal, elements contained in them must be equal and in the same order. We have found some cases in JDK 1.5 where array comparison is not done in full and that have some domain specific semantics. For instance, let's consider the *java.util.BitSet* class:

```
public class BitSet implements ... {
  private long bits[];
  private transient int unitsInUse = 0;

  public boolean equals(Object obj) {
    ...
    BitSet set = (BitSet) obj;
    // Get minimum bits used by both this and that
    int minUnitsInUse = Math.min(unitsInUse, set.unitsInUse);
    for (int i = 0; i < minUnitsInUse; i++)
      if (bits[i] != set.bits[i]) return false;
    // Check any bits in use by only one BitSet (must be 0 in other)
    if (unitsInUse > minUnitsInUse) {
      for (int i = minUnitsInUse; i<unitsInUse; i++)
        if (bits[i] != 0) return false;
    } else {
      for (int i = minUnitsInUse; i<set.unitsInUse; i++)
        if (set.bits[i] != 0) return false;
    }
    return true;
}
```

The number of bits used by each *BitSet* is defined by the *unitsInUse* field. So, a minimum on *this.unitsInUse* and *set.unitsInUse* will give the bit length that is used by both. The

```
public class CMStateSet {
  public boolean equals(Object o) {
    if (!(o instanceof CMStateSet)) return false;
    return isSameSet((CMStateSet)o);
  }
  final boolean isSameSet(CMStateSet setToCompare) {
    if (fBitCount != setToCompare.fBitCount) return false;
    if (fBitCount < 65) {
      return ((fBits1 == setToCompare.fBits1) && (fBits2 == setToCompare.fBits2));
    }
    for (int index = fByteCount - 1; index >= 0; index--) {
      if (fByteArray[index] != setToCompare.fByteArray[index]) return false;
    }
    return true;
  }
}
```

Figure 4.15: Equals of the CMStateSet class.

first *for-loop* checks if the bit value matches for both *this* and *set* for the common used length. If that matches, the following *if* and *else* block just check whether the extra used bits either on *this* or set are disabled. If an excess bit is not disabled then two bit sets will not be equal.

The bit array in this case has a domain specific representation and is not general enough to be supported by the checker. The checker expects *bits[i] != set.bits[i]* instead of a *bits[i] != 0* comparison. Instead of a bit array of *long*, it could have been a *char* array and instead of checking for a specific value like *0* for excess bits, a character *A* could have been used and abstraction of such a pattern would be difficult if not impossible. We found *6* cases of varying domain specific representations in array fields from JDK 1.5 and *1* case from Luence 3.0 that are not supported by the checker.

4.6.8 Domain specific representation on field

The Alloy model for state comparison abstracts the type of state being compared to an integer type and expects *this.state* compared to *that.state* using equality operators (== or *!=*) or equality methods (*equals* or *compareTo*). Given such a scenario, if a field is not compared between *this* and *that* objects, the checker cannot translate it to Alloy code. For instance, let's consider the *com.sun.org.apache.xerces.internal.impl.dtd.models.CMStateSet* class from JDK 1.5 in Figure 4.15.

The *CMStateSet* class is a lightweight version of the *java.util.BitSet* class. Its *equals*

method calls the *isSameState* method where states of *this* and *that* are compared. The *CMStateSet* class, however, has two internal representation: one for bit length *less than or equal* to *64* bits and another for *greater than 64*. The *if* statement with the $fBitCount < 65$ condition just switches between the representation to be used for equality comparison. However, the checker does not know of this domain specific comparison and expects *this.fBitCount* to be compared to *that.fBitCount* either with the == or *!*= operator. Hence, due to the field comparison with a constant and use of the non equality operator, the checker cannot translate it to an Alloy code. We have altogether *9* similar cases in JDK 1.5 and *1* case in Lucene 3.0 that are not supported by the checker.

4.6.9 Comparison delegated to a field

The equals checker is a hierarchy based analysis tool. We only expand method calls on *this* or *that* object but not on fields. When the helper method for equality comparison is implemented on a field, such a call cannot be expanded. For instance let's consider the *equals* method of *java.io.File* class from JDK 1.5:

```
public class File implements Serializable, Comparable<File> {
  static private FileSystem fs = FileSystem.getFileSystem();
  public boolean equals(Object obj) {
    if ((obj != null) && (obj instanceof File)) {
    return compareTo((File)obj) == 0;
    }
    return false;
  }

  public int compareTo(File pathname) {
    return fs.compare(this, pathname);
  }
}
```

In the code snippet, the *equals* method delegates equality comparison to the *compareTo* method. Inside the *compareTo* method *this* and *that* (pathname) objects are passed to the compare method of static filed *fs* of type FileSystem. The *fs* field represents a local operating system specific file system in which two files are compared for equality. The

checker, however, does not expand the *fs.compare* method and cannot reason about its correctness. We have *9* similar cases from JDK 1.5 that are not supported by the checker.

4.6.10 Polymorphic field

Sometimes, a type of field is not concrete and is abstracted as an *Object* type. This introduces polymorphism in the possible types for a field. In such a scenario, the field is type checked inside the *equals* method before performing relevant comparison. For instance, let's consider the *com.sun.jndi.ldap.SimpleClientId* class from JDK 1.5:

```
class SimpleClientId extends ClientId {
  final private String username;
  final private Object passwd;
  public boolean equals(Object obj) {
    ...
    SimpleClientId other = (SimpleClientId) obj;
    return (... && ((passwd == other.passwd)
      || (passwd != null && other.passwd != null &&
         (
          ((passwd instanceof String) && passwd.equals(other.passwd)) ||
          ((passwd instanceof byte[]) && (other.passwd instanceof byte[]) &&
          Arrays.equals((byte[]) passwd, (byte[]) other.passwd)) ||
          ((passwd instanceof char[]) && (other.passwd instanceof char[]) &&
          Arrays.equals((char[]) passwd, (char[]) other.passwd))
          )));
  }
}
```

SimpleClientId represents an identity of an authenticated *LDAP* connection. This class supports authentication through *username* and *password* represented by the *username* and *passwd* field. The field *username* is by default *String* type however the class relaxes the type constraint on the *passwd* field by making it *Object* type. Inside the *equals* method, the runtime type of *passwd* is checked before calling the appropriate equality method. In this case, the types of fields are limited to be either *String*, *byte[]* or *char[]* arrays. Since the checker abstracts the type of field to be an integer type in the Alloy model, such type checking statements cannot be translated to Alloy. We have *2* similar cases not supported

by the checker from JDK 1.5.

4.6.11 Creation of new state for equality

A state in an *equals* method is generally compared between *this* and *that* without any modification. If a state is modified by creating a new local state either by cloning or by instantiating a new local object through a parametrized constructor, then the checker cannot reason about such comparison. For instance, let's consider the *sun.security.x509.KeyIdentifier* class where cloning is used:

```
public class KeyIdentifier {
  private byte[] octetString;
  public byte[] getIdentifier() { return ((byte[])octetString.clone()); }

  public boolean equals(Object other) {
    if (this == other) return true;
    if (!(other instanceof KeyIdentifier)) return false;
    return java.util.Arrays.equals(octetString, ((KeyIdentifier)other).getIdentifier());
  }
}
```

In the code snippet, the *this.octetString* byte array is compared to the clone of *other.octetString*. Cloning may create a new local object and cloned objects are not necessarily equal. Hence, such comparison where *this.octetString* is compared to a cloned local array is not handled by the checker. Nevertheless, for this particular problem we found cloning unnecessary because *Arrays.equals([],[])* does not mutate its parameters.

Another example where a new operator is used along with a parameterized constructor to create a new local object for comparison can be found in the *javax.security.auth.Subject* class from JDK 1.5 shown in Figure 4.16.

The principals field in the class is a synchronized set as we can see in the constructor. If a mere comparison of *this.principals.equals(that.principals)* was done in the *equals* method instead of the above code snippet then there was a possibility of a deadlock. This may happen when *this.pricipals* and *that.principals* have different mutex variables (or locks) and two threads are simultaneously trying to perform an equality check on *principals*; one doing

```
public class SslRMIServerSocketFactory implements RMIServerSocketFactory {
  Set principals;
  public Subject() { this.principals = Collections.synchronizedSet(...); ... }

  public boolean equals(Object o) {
    final Subject that = (Subject)o;
    Set thatPrincipals;
    synchronized(that.principals) {
      // avoid deadlock from dual locks
      thatPrincipals = new HashSet(that.principals);
    }
    if (!principals.equals(thatPrincipals)) { return false; }
    ...
  }
}
```

Figure 4.16: Equals of the SslRMIServerSocketFactory class.

this.principal.equals(that.principal) and the other doing *that.principal.equals(this.principal)*. To avoid deadlock due to this dual lock, the *that.principals* field is un-synchronized by initializing a new local *HashSet* object out of it. The un-synchronized *HashSet*(*thatPrincipals* in the code snippet) is then compared to *this.principals* which is effectively comparing the value of both sets for equality. Due to the equality comparison of *this.principals* with the new local object, the checker cannot translate such a coding pattern to Alloy. In fact, it does not know if *thatPrincipals* is semantically equal to *that.principals*. We have *11* similar cases from JDK 1.5 that is not supported by the checker.

4.6.12 Wrapped state Comparison

```
private static class CheckedEntry<K,V> implements Map.Entry<K,V> {
  private Map.Entry<K, V> e;

  public boolean equals(Object o) {
    if (!(o instanceof Map.Entry)) return false;
    Map.Entry t = (Map.Entry)o;
    return eq(e.getKey(),   t.getKey()) && eq(e.getValue(), t.getValue());
  }
}
// Defined in Collections class
private static boolean eq(Object o1, Object o2) {
  return (o1==null ? o2==null : o1.equals(o2));
}
```

Figure 4.17: Equals of the CheckedEntry class.

The wrapper pattern has already been explained in Section 4.5.13. Sometimes instead of performing *wrapped.equals(that)*, some specific field of the wrapped object are used for state comparison. For instance, let's consider the *java.util.Collections.CheckedMap*

.CheckedEntrySet.CheckedEntry class from JDK 1.5 (Figure 4.17).

In the code snippet, *key* and *value* of the wrapped object *e* are compared instead of comparing wrapped object *e* with *o* with *e.equals(o)* as in the regular Wrapper pattern. There are altogether *2* similar cases in JDK 1.5 that are not supported by the checker.

4.6.13 Domain specific equality

It is good to check if *this* and *that* represent the same physical address by performing a *this == that* check which is called *identity* or *reference* equality. We have found a case in JDK 1.5 where *this* is compared to a constant object in the *equals* method. It's in the *java.security.spec.ECPoint* class:

```
public class ECPoint {
  public static final ECPoint POINT_INFINITY = new ECPoint();
  private final BigInteger x;
  private final BigInteger y;

  // private constructor for constructing point at infinity
  private ECPoint() {
    this.x = null;
    this.y = null;
  }

  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (this == POINT_INFINITY) return false;
    if (obj instanceof ECPoint) {
      return ((x.equals(((ECPoint)obj).x)) && (y.equals(((ECPoint)obj).y)));
    }
    return false;
  }
}
```

The *ECPoint* class represents a point in an elliptic curve in an affine coordinate system. The default constructor creates a point at infinity represented by both coordinates x and y set to null. The *POINT_INFINITY* constant is instantiated to represent a point at infinity. When it comes to performing the equality check for such an object, it makes sense to make it

Description	Tomcat 6	Lucene 3.0	JDK 1.5
Equals search	5s (22%)	4s (12%)	1m 7s (10%)
Hierarchy construction	<1s (3%)	2s (6%)	21s (3%)
Path generation	2s (9%)	2s (7%)	3m 12s (29%)
Alloy model generation	13s (52%)	6s (19%)	5m 20s (49%)
Alloy model checking	3s (12%)	18s (53%)	41s (6%)
Total time	25s	33s	10m 42s

Table 4.6: Summary of execution time.

not equal to everything else. This is a very domain specific implementation for non-equality and is not supported by the checker.

4.6.14 Execution Time

Table 4.6 shows the time taken for each phase in the analysis of the three projects. The machine used for analyses is a MacBookPro Laptop with a 2.4 GHz Intel Core 2 Duo processor and 2 GB of main memory. The Java Virtual Machine was given 800 MB of minimum, and 1024 MB of maximum memory through the Eclipse IDE for analysis. Our experiments also confirmed that the minimum memory of 400 MB worked for Lucene and Tomcat, and 600 MB for JDK. The Eclipse IDE itself used about 250 MB.

The performance data looks very promising, indicating that this technique can scale up to a project as large as JDK 1.5, and, thus, can probably be used on a daily basis on a developer's desktop. In general, the checker has a time and space trade-off: the more the memory, the faster the analysis. With less memory, the checker resets Soot more often before loading a type hierarchy, introducing time overhead.

Chapter 5

Related Work

5.1 Equals Design and Implementations

Liskov and Guttag suggest to use reference equality for mutable objects and value equality for immutable objects [30]. For example, an *IntSet* in their work uses reference equality. In contrast, many data types in Java Collection Framework [9] employ value based equality. In fact, Java's specification for *equals()*, in particular, the consistency rule, is more relaxed and allows a mutable class to define *equals()*. The tradeoff is that the developer must ensure that the consistency rule is obeyed throughout his application.

When a type-compatible equality is implemented between a supertype and a subtype, their *equals()* conform to LSP. When two types implement a type-incompatible equality or a hybrid equality, they may violate LSP if specifications other than the default equals contract are assumed. Some (e.g., Bloch [10]) suggest to use composition instead of inheritance in such cases. We believe that type-incompatible equality should not be the main reason for two classes to not have a subtyping relation, and that it can still be useful for two types to subtype each other when their *equals()* violate LSP. In contrast to [10], the hybrid equality shows that it is possible to extend an instantiable class and add an aspect while preserving the equals contract.

The implementation for the hybrid equality essentially uses a template method design

pattern [22]. Use of the template method design pattern for implementing *equals()* has also been proposed by Cohen [12, 40] and later by Stevenson and Phillips [41]. Our work is broader than theirs in that we provide a set of guidelines that is aimed to cover all of the relevant design and implementation considerations. Unlike the analytical work in [12, 41], we conducted an empirical study with real-life projects, with findings that both validate and enrich our guidelines. Furthermore, our design and implementation of hybrid equality appears to be simpler than theirs and can be more familiar to programmers for a quick start.

Baker [7] attempts to define a semantics model for object equality in the context of Lisp. However, the goal seems to be to completely implement equality at the programming language level. We believe that equality is domain-specific and requires developer involvement in its definition.

5.2 Automated equals() Code Generation

Vaziri et al. [45] extend Java with relation types with which equality can be specified in terms of object properties and the *equals()* implementation can be generated automatically. However, the relation types neither take into account the Object's behaviors for equality comparison, nor support type-compatible and hybrid equality (the generated implementation uses *getClass()* only). Therefore, it would help solve only a subset of *equals()* implementation issues.

IDEs such as Eclipse [17] and NetBeans [35] have wizards that generate *equals()*. NetBeans 6.8 allows for only *getClass()* and fails to include the state of a superclass into the *equals()* of a subclass. Eclipse 3.5 gives a developer an option to choose from *instanceof* and *getClass()* but does not support the implementation of *hybrid* equality.

Rayside et al. use annotations to specify an abstract view of an object's representation that helps to automatically generate *equals()* code [36]. They use Java Collection Framework as one of their benchmarks where they modify existing source code to add relevant annotations. Our approach does not change existing source code. We have several pattern

detectors that automatically detect the abstraction functions in the implementation of the *equals()* method thus reducing an extra burden on the programmer. Furthermore, it is not clear from the paper if they can automatically generate all of the different equals implementation patterns that our detector can handle. One simple example is the *equals()* method of the *org.apache.lucene.search.Query* class from Apache Lucene 3.0:

```
public abstract class Query implements ... {
  private float boost = 1.0f;                    // query boost factor

  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;
    Query other = (Query) obj;
    if (Float.floatToIntBits(boost) != Float.floatToIntBits(other.boost)) return false;
    return true;
  }
}
```

Float.floatToIntBits() returns an integer representation of the float value according to the IEEE 754 floating-point single format bit layout. We found 38 sub types of Query class that directly or indirectly rely on such state check pattern where instead of checking the field directly, the *equals()* method checks certain properties of the field for equality. It is unclear how 5 different annotations presented in their paper can handle such equality patterns. Furthermore, it is also not clear how a *hybrid* equality pattern can be generated using these annotations.

5.3 Program Analysis and Comprehension

Soot provides a 3-address based intermediate representation of Java code called Jimple [44]. Jimple is also a stack-less representation and has a smaller number of instructions to be handled than Java bytecode. Nevertheless, its flow-analysis framework is built on top of intra-procedure control-flow graphs that do not suffice our need. We extend Soot to provide an inter-procedure path-based flow analysis framework. Path-based analysis, in general, is

more precise than flow graph analysis as flow sets do not have to be merged in a path. Flow graphs based program analyses, have to merge flow sets when two control flow branches meet together, thus, introducing inaccuracy.

Closely related work to our inter-procedure analysis is pointer analysis. A very simple approach to pointer analysis is not to use context information at all. Steensgaard proposes an efficient pointer analysis algorithm which is context insensitive, flow insensitive, unification based. Spark [29] points to analysis framework that comes with Soot is also inclusion based. Similarly, Whale and Lam propose field-sensitive (object and its field are treated as different memory locations) and *intra-procedure* flow sensitive analyses [46]. Our inter-procedure analysis is context and flow sensitive. Since, our approach is path sensitive as well, we can query for pointer alias with respect to a path, thus, getting a more accurate answer. Other context sensitive but flow insensitive analyses can be found in [21, 19].

A flow-insensitive but object-sensitive analysis is proposed by Milanova et al. [34]. In their approach, a method is analyzed separately for each of the object names that represent run-time objects on which that method may be invoked. Like their approach we also analyze methods that may be invoked on *this* or *that* receiver objects but we have flow-sensitivity in our analyses that they do not have.

Shivers proposes *k-CFA* where only k call sites are remembered for limiting the number of contexts in control flow analysis [39]. This is typically useful for recursive calls. We expand a recursive call only once which is conceptually similar to [18]. Furthermore, our context sensitivity is clone-based, similar to [18, 47]. But, unlike [47] we have flow-sensitivity at all points in a path. The key difference between all of the mentioned analyses and ours is that we apply inter-procedure analysis for a specific problem domain, i.e. *equals()* method checking. Hence, our entry method is not a *main()* method or a *Thread.run()* method but an *equals()* method. Furthermore, the *equals()* method generally does not initialize new objects, instead, it has logic for object comparisons. Hence, our pointer analysis is more like a reference sensitive analysis where we query which local variable points to the *this* object or a field of the *this* object (or parameter of the *equals()* method). Nevertheless, we

can also answer an initialization related query if an object is initialized at a reachable point from the *equals()* method.

All of the inter-procedure analyses need an access to a call graph generally with *main()* or *Thread.run()* methods as an entry method. For instance, [46, 47] needs access to pre-computed call graphs. Like [19] we do not require a pre-computed call graph and evaluate on-the-fly targets. We run context-senstive type analysis on the receiver object along with a class hierarchy analysis [14] to compute possible dispatch targets. Since, our entry method is *equals()*, chances are the object is not initialized in its paths. In such scenarios, we rely on the type checking statements (use of *instanceof*, *getClass()*, type casting, etc) within the program body for better precession. In fact, our *type checking filter* significantly prunes invalid paths during path generation (see Section 4.3).

An intra-procedurally path-sensitive and summary-based, context-sensitive program analysis framework is developed for C in [3]. This framework is extended in [15] to evaluate several system constraints inter-procedurally. An example of such a constraint is null pointer dereferencing in C. In a modular program analysis (program analyzed in several parts), they introduce a concept of unobservable variable for values that are not known statically. They model such a variable in terms of existential quantifiers which we do through domain specific abstraction whenever possible. Das et al. propose a path-sensitive program verification tool called ESP [13] for the C language. Their approach is similar to ours in a way that their path based analysis is governed by domain semantics. They try to track only relevant branches pertaining to the property to be checked which they also call inter-procedural property simulation.

Automatic code recognition has been a challenge for program comprehension for a long time. Johnson and Soloway have developed a framework called PROUST for on-line analysis and understanding of Pascal programs written by novice programmers [28]. They take program requirements as an input, identify several functional units in the program and a set of plans to be executed while writing the program. They check if the plans have been successfully executed by determining the correctness of the functional units. Our functional

units are the comparison patterns that we check for correctness.

Seemann and Gudenberg propose a pattern-based design recovery technique using call graphs [38]. They are primarily interested in detecting design patterns related to inheritance and composition. We also, for instance, detect Adapter patterns in the *equals()* code by investigating the method call site and composing the type of the receiver object (e.g. *field*, *this*, *that*, etc). Their approach of relying on some well-defined name for abstraction is similar to us. For instance, we do not expand the *int hashCode()* method. However, our pattern detectors are not limited to names, inheritance or composition. We perform abstraction at several levels, variable name, composite statement(s) (combination of one or more statements and their execution sequence to denote certain comparisons like array patterns), inheritance where we look for subtypes of collection interfaces for particular patterns (e.g. List, Set and Map). Hence, we look at both structural as well as behavioral properties in code to perform abstraction at various level. A query by outlines paradigm is introduced by Balmas in a prototype tool called QBO [8]. It can answer queries based on constraints imposed on an outline model. For instance, a query can be *where a variable is modified*? We use flow analysis to accurately answer such questions.

Design and implementation of Alloy models are discussed in details in [27]. However, the equivalence relation in particular is not discussed. Marinov et al. propose VAlloy (an extension to Alloy), that models virtual functions [32]. In their work, models of different Java classes are discussed in VAlloy. We automate the code generation process and directly generate the code in Alloy. VAlloy code has to be written manually which is then translated to Alloy and then run for error checking. Nevertheless, the representation of VAlloy is more compact than Alloy for virtual function modeling.

5.4 Existing Checkers

There are several static checkers available for Java. Among them FindBugs [43] handles more cases related to *equals()* than others. It detects 36 *equals()* related problems most of which are related to *equals() use* rather than its *implementation*. The 4 categories of

problems that are related to equivalence relation detected by the tool are:

Equals checks for non-compatible operand This is the same as overloading equals implementations for multiple purposes discussed in Section 2.2.2.

Equals method always returns false The tool checks if an *equals* method always returns false. This will violate the reflexive property of the equivalence relation.

Equals method always returns true In this case the tool checks if an *equals* method always returns true. This may violates the symmetric property of the equivalence relation.

Equals method overrides equals in superclass and may not be symmetric This is very similar to suspicious implementations of type-incompatible equality discussed in Section 2.2.3. In particular, FindBugs looks for the *instanceof* operator in the *equals()* method of both the super-type and sub-type to detect symmetric property violations. This is very unreliable checking and will miss a lot of potential problems that may occur without the use of the *instanceof* operator.

FindBugs does not address issues related to hybrid equality or proper implementation of *equals* as proposed in our guidelines. It does not detect all of the symmetric and transitive property violations detected by our checker.

Hou et al. [24, 23] use SCL to specify and detect violation of implementation constraints for *equals() within each individual class*, but inheritance is not the focus of that work.

Flanagan et al. propose a model checker for Java that relies on programmers' annotations for checking constraints [20]. Nevertheless, they do not focus on the equivalence relation.

Chapter 6

Conclusion

6.1 Future Work

A very near future work is to extend the checker to handle all of the supportable unknown code patterns for better code coverage. Our Alloy model mostly focuses on inheritance and research on providing better support for composition can be another direction. Testing with VAlloy [32] models can be an alternative.

The SERL Framework is a useful program analysis tool and can have several applications. One such application can be detecting domain specific constraints on a framework. We have investigated several scattered concerns for a JTree API in [25]. These concerns are also common to most of the Java Swing APIs. It is possible to implement pattern detectors for these concerns and provide suggestions to the programmers when constraints related to any concern is violated. This will effectively give rise to a dynamic help system for the programmers that can provide context related expert suggestions for using framework code.

Another direction is to mature a SERL Framework into a demand-driven inter-procedure path-sensitive context-sensitive analysis framework. Demand-driven in the sense unlike most other inter-procedure analysis where call graphs are pre-computed, we will provide APIs for selectively processing method expansion based on heuristics and proper abstraction.

6.2 Conclusion

We have shown that the implementation of a seemingly simple method like *equals()* can cause many problems. It is a general design problem that also appears in other languages like C#. Fortunately, its solution goes hand in hand with the established OO design methods and principles such as behavioral subtyping. In particular, the identification of design intent for the right equality is crucial for the proper implementation of *equals()*. Furthermore, our analysis of several open source projects shows that this contract is widely depend on and it can be easily implemented improperly. Our analysis of the root causes for *equals()*-related problems provides insights and helps create concrete guidelines on things that should be done and things that should be avoided. These guidelines will be useful not only to software developers but also tool vendors, since modern IDEs like Eclipse and NetBeans still cannot generate problem-free *equals()*. We have further applied these ideas on a model-based static checker that was able to detect a majority of the problems. Furthermore, documentation of the cases not supported by the checker will provide a starting point for future research.

In general, it appears to us that the equality design is a representative example for design extension, a software development style that is increasingly becoming common in frameworks and other reuse-based development. Proper design extension requires the developers to possess enough knowledge about the design, who often lack such knowledge and as a result, write suboptimal code. We plan to investigate design guidelines for other extensible designs, and to develop tools to better assist the developers in the design extension process.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006, pp. 583–705.

[2] ——, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006, pp. 903–964.

[3] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the saturn project," in *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM, 2007, pp. 43–48.

[4] Apache Lucene. The Apache Software Foundation. Last verified: April 18, 2010. [Online]. Available: http://lucene.apache.org/java/docs

[5] Apache Tomcat. The Apache Software Foundation. Last verified: April 18, 2010. [Online]. Available: http://tomcat.apache.org/

[6] BCEL. The Apache Software Foundation. Last verified: April 18, 2010. [Online]. Available: http://jakarta.apache.org/bcel

[7] H. G. Baker, "Equal Rights for Functional Objects or, the More Things Change, the More They Are the Same," *SIGPLAN OOPS Mess.*, vol. 4, no. 4, pp. 2–27, 1993.

[8] F. Balmas, "Query by outlines: a new paradigm to help manage programs," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 5, pp. 86–94, 1999.

[9] J. Bloch. Java Collections Framework. Last verified: April 18, 2008. [Online]. Available: http://java.sun.com/docs/books/tutorial/collections/index.html

[10] ——, *Effective Java programming language guide.* Mountain View, CA, USA: Sun Microsystems, Inc., 2001.

[11] J. Bloch and N. Gafter, *Java Puzzlers: Traps, Pitfalls, and Corner Cases.* Addison-Wesley Professional, 2005.

[12] T. Cohen, "How Do I Correctly Implement the equals() Method?" *Dr. Dobb's Journal*, May 2002.

[13] M. Das, S. Lerner, and M. Seigle, "Esp: path-sensitive program verification in polynomial time," in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation.* New York, NY, USA: ACM, 2002, pp. 57–68.

[14] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming.* London, UK: Springer-Verlag, 1995, pp. 77–101.

[15] I. Dillig, T. Dillig, and A. Aiken, "Sound, complete and scalable path-sensitive analysis," *SIGPLAN Not.*, vol. 43, no. 6, pp. 270–280, 2008.

[16] JDT: Java Development Tools. The Eclipse Foundation. Last verified: April 18, 2010. [Online]. Available: http://www.eclipse.org/jdt

[17] The Eclipse Project. The Eclipse Foundation. Last verified: April 18, 2010. [Online]. Available: http://www.eclipse.org/eclipse

[18] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *PLDI '94: Proceedings of the*

ACM SIGPLAN 1994 conference on Programming language design and implementation. New York, NY, USA: ACM, 1994, pp. 242–256.

[19] M. Fähndrich, J. Rehof, and M. Das, "Scalable context-sensitive flow analysis using instantiation constraints," *SIGPLAN Not.*, vol. 35, no. 5, pp. 253–263, 2000.

[20] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. New York, NY, USA: ACM, 2002, pp. 234–245.

[21] J. S. Foster, M. Fähndrich, and A. Aiken, "Polymorphic versus monomorphic flow-insensitive points-to analysis for c," in *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*. London, UK: Springer-Verlag, 2000, pp. 175–198.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[23] D. Hou, "SCL: Static Enforcement and Exploration of Developer Intent in Source Code," in *ICSE'07 COMPANION*, 2007, pp. 57–58.

[24] D. Hou and H. J. Hoover, "Using SCL to Specify and Check Design Intent in Source Code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, pp. 404–423, 2006.

[25] D. Hou, C. Rupakheti, and H. Hoover, "Documenting and evaluating scattered concerns for framework usability: A case study," in *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, dec. 2008, pp. 213 –220.

[26] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[27] ——, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[28] W. L. Johnson and E. Soloway, "Proust: Knowledge-based program understanding," *IEEE Trans. Softw. Eng.*, vol. 11, no. 3, pp. 267–275, 1985.

[29] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," in *Compiler Construction, 12th International Conference*, ser. LNCS, G. Hedin, Ed., vol. 2622. Warsaw, Poland: Springer, April 2003, pp. 153–169.

[30] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[31] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1811–1841, 1994.

[32] D. Marinov and S. Khurshid, "Valloy - virtual functions meet a relational language," in *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right.* London, UK: Springer-Verlag, 2002, pp. 234–251.

[33] C# Language Specification. Microsoft Inc. Last verified: April 18, 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa645596(VS.71).aspx

[34] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 1, pp. 1–41, 2005.

[35] The NetBeans IDE. Netbeans.org. Last verified: April 18, 2010. [Online]. Available: http://www.netbeans.org

[36] 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," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 342–352.

[37] C. R. Rupakheti and D. Hou, "An empirical study of the design and implementation of object equality in java," in *CASCON '08: Proceedings of the 2008 conference of the*

center for advanced studies on collaborative research. New York, NY, USA: ACM, 2008, pp. 111–125.

[38] J. Seemann and J. W. von Gudenberg, "Pattern-based design recovery of java software," in *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering.* New York, NY, USA: ACM, 1998, pp. 10–16.

[39] O. G. Shivers, "Control-flow analysis of higher-order languages of taming lambda," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[40] R. Smit. Introducing equivalent types in equals() implementation. Last verified: April 18, 2008. [Online]. Available: http://www.forum2.org/tal/equals.html

[41] D. E. Stevenson and A. Phillips, "Implementing Object Equivalence in Java Using Template Method Design Pattern," in *SIGCSE 2003 Technical Symposium on Computer Science Education*, Reno, Nevada, USA, Feb. 2003, pp. 278–282.

[42] JDK 5.0 Documentation. Sun Microsystems. Last verified: April 18, 2010. [Online]. Available: http://java.sun.com/j2se/1.5.0/docs

[43] FindBugs. The University of Maryland. Last verified: April 18, 2010. [Online]. Available: http://findbugs.sourceforge.net/

[44] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research.* IBM Press, 1999, pp. 125–135.

[45] M. Vaziri, F. Tip, S. Fink, and J. Dolby, "Declarative Object Identity Using Relation Types," in *21st European Conference on Object-Oriented Programming*, Berlin, Germany, Aug. 2007, pp. 54–78.

[46] J. Whaley and M. S. Lam, "An efficient inclusion-based points-to analysis for strictly-typed languages," in *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*. London, UK: Springer-Verlag, 2002, pp. 180–195.

[47] ——, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. New York, NY, USA: ACM, 2004, pp. 131–144.

Appendices

Appendix A

Language Specification

Language Specification for Alloy Model in Extended Backus-Naur Form:

```
<Module> ::=
  <ModuleDeclaration>
  <TypeDeclarations>
  <EqualityPredicate>
  <TypeExclusion>
  <EquivalenceSpecification>
  <ValidityCheck>

<ModuleDeclaration> ::= "" | "module" .+

<TypeDeclarations> ::=
  <TypeDeclaration> | <TypeDeclaration> <TypeDeclarations>

<TypeDeclaration> ::=
  { "abstract" | "" } "sig" <Id> <Inheritance> "{" <FieldDeclarations> "}" <FactSpecifications>

<Id> ::= [_A-Za-z][_A-Za-z0-9]*

<Inheritance> ::= "" | "extends" <Id> | "in" <Id>

<FieldDeclarations> ::=
  <FieldDeclaration> | <FieldDeclaration> "," <FieldDeclarations>

<FieldDeclaration> ::= <Id> ":" <Type>

<Type> ::=  "Int" | "seq Int" | "set Int" | "Int -> Int"
```

```
<FactSpecifications> ::= "" | <FactSpecification> <FactSpecifications>

<FactSpecification> ::= "fact" "{" <Id> "in" <Id> "}"

<EqualityPredicate> ::=
  "pred" <Id> "::" "equals" "(" <Id> ":" <Id> ")" "{"
    <Implications>
  "}"

Implications ::=
  <Implication> | <Implication> "else" <Implications>

<Implication> ::=
  "(" <Id> "in" <TypeIsolation> ")" "=>" <FactBlocks>

<TypeIsolation> ::= <Id> | <Id> "-" TypeIsolation

<FactBlocks> ::=
  <FactBlock> | <FactBlock> "or" <FactBlocks>

<FactBlock> ::=
  <Fact> | <Fact> "and" <FactBlock>

<Fact> ::=
  <TypeCheck> | "!" <TypeCheck>
  | <StateCheck> | "!" <StateCheck>

<TypeCheck> ::= "(" <Id> "in" <TypeIsolation> ")"

<StateCheck> ::=
  "(" <Id> "=" <Id> ")"
  | "(" <Id>"."<Id> "=" <Id>"."<Id> ")"

<TypeExclusion> ::= <TypeDeclaration>

<EquivalenceSpecification> ::=
"assert reflexive { all a : " <TypeIsolation> "| a.equals[a] }"
"assert symmetric { all a, b : " <TypeIsolation> "| a.equals[b] <=> b.equals[a] }"
"assert transitive { all a, b, c : " <TypeIsolation> "| a.equals[b] and b.equals[c] => a.equals[c] }"
```

```
<ValidityCheck> ::=
  "check reflexive for 1"
  "check symmetric for 2"
  "check transitive for 3"
```

Appendix B

Reported Errors

Note that false positives are marked with a *****. Also note that **@ num** means problematic pattern starts at **num** line number.

Reflexive Property Violation

JDK 1.5

1. com.sun.tools.javac.util.Paths.PathIterator
2. java.net.InetAddress
3. sun.management.MappedMXBeanType.InProgress
4. com.sun.jmx.snmp.IPAcl.GroupImpl*
5. javax.management.openmbean.OpenMBeanParameterInfoSupport*

Symmetric Property Violation

JDK 1.5

1. java.util.AbstractList (Root of type hierarchy)
   - com.sun.corba.se.spi.ior.IdentifiableContainerBase
   - com.sun.corba.se.impl.ior.iiop.IIOPProfileTemplateImpl
2. com.sun.jmx.snmp.IPAcl.PrincipalImpl
   - com.sun.jmx.snmp.IPAcl.GroupImpl

- com.sun.jmx.snmp.IPAcl.GroupImpl

3. com.sun.org.apache.xerces.internal.xni.QName
   - com.sun.org.apache.xerces.internal.xni.QName
   - com.sun.org.apache.xerces.internal.impl.dv.xs.QNameDV.XQName
4. com.sun.security.auth.NTSid
   - com.sun.security.auth.NTSid
   - com.sun.security.auth.NTSidUserPrincipal
5. com.sun.tools.apt.mirror.declaration.DeclarationImpl
   - com.sun.tools.apt.mirror.declaration.ParameterDeclarationImpl
   - com.sun.tools.apt.mirror.declaration.ClassDeclarationImpl
6. java.awt.geom.Rectangle2D
   - java.awt.geom.Rectangle2D.Double
   - javax.swing.text.DefaultCaret
7. javax.imageio.ImageTypeSpecifier
   - javax.imageio.ImageTypeSpecifier.Grayscale
   - javax.imageio.ImageTypeSpecifier.Banded
8. javax.management.MBeanOperationInfo
   - javax.management.MBeanOperationInfo
   - javax.management.openmbean.OpenMBeanOperationInfoSupport
9. javax.naming.RefAddr
   - javax.naming.StringRefAddr
   - javax.naming.BinaryRefAddr
10. javax.security.auth.kerberos.ServicePermission
    - javax.security.auth.kerberos.ServicePermission
    - javax.security.auth.kerberos.ServicePermission

Lucene 3.0

1. org.apache.lucene.search.Query (Root of type hierarchy)
    - org.apache.lucene.search.spans.SpanNearQuery
    - org.apache.lucene.search.payloads.PayloadNearQuery

Tomcat 6

1. java.util.AbstractMap (Root of type hierarchy)
    - org.apache.catalina.tribes.tipis.ReplicatedMap
    - java.util.concurrent.ConcurrentHashMap

Transitive Property Violation

JDK 1.5

1. java.util.AbstractList (Root of type hierarchy)
    - com.sun.corba.se.impl.ior.FreezableList
    - java.util.concurrent.CopyOnWriteArrayList.COWSubList
2. com.sun.jmx.snmp.IPAcl.PrincipalImpl
    - com.sun.jmx.snmp.IPAcl.GroupImpl
    - com.sun.jmx.snmp.IPAcl.GroupImpl
3. com.sun.jndi.ldap.LdapName.TypeAndValue
    - com.sun.jndi.ldap.LdapName.TypeAndValue
    - com.sun.jndi.ldap.LdapName.TypeAndValue
4. com.sun.jndi.ldap.NotifierArgs*
    - com.sun.jndi.ldap.NotifierArgs
    - com.sun.jndi.ldap.NotifierArgs
5. com.sun.org.apache.xerces.internal.xni.QName
    - com.sun.org.apache.xerces.internal.xni.QName
    - com.sun.org.apache.xerces.internal.impl.dv.xs.QNameDV.XQName
6. com.sun.tools.apt.mirror.declaration.DeclarationImpl
    - com.sun.tools.apt.mirror.declaration.ParameterDeclarationImpl

- com.sun.tools.apt.mirror.declaration.TypeParameterDeclarationImpl

7. java.util.ResourceBundle.ResourceCacheKey*
   - java.util.ResourceBundle.ResourceCacheKey
   - java.util.ResourceBundle.ResourceCacheKey
8. javax.imageio.ImageTypeSpecifier
   - javax.imageio.ImageTypeSpecifier.Packed
   - javax.imageio.ImageTypeSpecifier.Banded
9. javax.management.MBeanOperationInfo
   - javax.management.MBeanOperationInfo
   - javax.management.openmbean.OpenMBeanOperationInfoSupport
10. javax.naming.RefAddr
    - javax.naming.StringRefAddr
    - javax.naming.BinaryRefAddr
11. javax.security.auth.kerberos.ServicePermission
    - javax.security.auth.kerberos.ServicePermission
    - javax.security.auth.kerberos.ServicePermission
12. sun.security.x509.X500Name
    - sun.security.x509.X500Name
    - sun.security.x509.X500Name

Lucene 3.0

1. org.apache.lucene.search.function.FieldCacheSource* (Root of type hierarchy)
   - org.apache.lucene.search.function.IntFieldSource
   - org.apache.lucene.search.function.IntFieldSource

Tomcat 6

1. java.util.AbstractMap (Root of type hierarchy)

- org.apache.catalina.tribes.tipis.LazyReplicatedMap
- java.util.concurrent.ConcurrentHashMap

Unguarded null parameter

JDK 1.5

1. com.sun.java.util.jar.pack.Package.File[@ 711]
2. com.sun.org.apache.xalan.internal.xsltc.compiler.FunctionCall.JavaType[@ 154]
3. com.sun.org.apache.xalan.internal.xsltc.compiler.VariableRefBase[@ 75]
4. com.sun.org.apache.xml.internal.dtm.ref.DTMNodeProxy[@ 121]
5. com.sun.org.apache.xml.internal.utils.CharKey[@ 82]
6. com.sun.org.apache.xpath.internal.objects.XString[@ 436]
7. com.sun.org.apache.xpath.internal.objects.XStringForFSB[@ 436]
8. java.awt.Font.Key[@ 663]
9. java.beans.ReflectionUtils.Signature[@ 279]
10. sun.security.x509.X509CertImpl[@ 1802]
11. sun.security.x509.X509CRLImpl[@ 1087]
12. java.util.Calendar[@ 2272]
13. sun.font.FontLineMetrics[@ 79]
14. sun.font.GlyphLayout.SDCache.SDKey[@ 266]
15. sun.rmi.rmic.iiop.CompoundType.Method[@ 1826]
16. sun.rmi.rmic.iiop.Type[@ 401]
17. sun.security.jgss.GSSNameImpl[@ 244]
18. sun.security.x509.X509Key[@ 427]
19. sun.swing.BakedArrayList[@ 66]
20. sun.text.IntHashtable[@ 77]

Lucene 3.0

1. org.apache.lucene.search.function.CustomScoreQuery[@ 140]

2. org.apache.lucene.search.function.OrdFieldSource[@ 104]
3. org.apache.lucene.search.function.ReverseOrdFieldSource[@ 114]
4. org.apache.lucene.search.function.ValueSourceQuery[@ 182]
5. org.apache.lucene.spatial.geometry.FixedLatLng[@ 148]
6. org.apache.lucene.spatial.geometry.FloatLatLng[@ 131]

Probable ClassCastException

JDK 1.5

1. com.sun.java.util.jar.pack.ConstantPool.ClassEntry[@ 325]
2. com.sun.java.util.jar.pack.ConstantPool.DescriptorEntry[@ 363]
3. com.sun.java.util.jar.pack.ConstantPool.MemberEntry[@ 422]
4. com.sun.java.util.jar.pack.ConstantPool.NumberEntry[@ 258]
5. com.sun.java.util.jar.pack.ConstantPool.SignatureEntry[@ 493]
6. com.sun.java.util.jar.pack.ConstantPool.StringEntry[@ 293]
7. com.sun.java.util.jar.pack.ConstantPool.Utf8Entry[@ 203]
8. com.sun.java.util.jar.pack.Package.File[@ 710]
9. com.sun.java.util.jar.pack.Package.InnerClass[@ 884]
10. com.sun.org.apache.xerces.internal.impl.dtd.XMLDTDDescription[@ 151]
11. com.sun.org.apache.xml.internal.utils.CharKey[@ 82]
12. java.awt.Font.Key[@ 662]
13. java.beans.ReflectionUtils.Signature[@ 278]
14. java.text.DecimalFormat[@ 1865]
15. java.text.PatternEntry[@ 56]
16. java.text.RuleBasedCollator[@ 732]
17. java.text.SimpleDateFormat[@ 1842]
18. sun.rmi.rmic.iiop.CompoundType.Method[@ 1824]
19. sun.rmi.rmic.iiop.Type[@ 401]

20. sun.swing.BakedArrayList[@ 63]

Lucene 3.0

1. org.apache.lucene.analysis.tokenattributes.PayloadAttributeImpl[@ 78]

Similarity for Equality

JDK 1.5

1. com.sun.corba.se.impl.orbutil.RepIdDelegator[@ 143]
2. com.sun.corba.se.impl.orbutil.RepIdDelegator_1_3[@ 147]
3. com.sun.corba.se.impl.orbutil.RepIdDelegator_1_3_1[@ 147]
4. com.sun.org.apache.xalan.internal.xsltc.compiler.FunctionCall.JavaType[@ 154]
5. com.sun.org.apache.xml.internal.utils.XMLStringDefault[@ 185]
6. com.sun.org.apache.xml.internal.utils.synthetic.reflection.Field[@ 94]
7. com.sun.org.apache.xpath.internal.Arg[@ 232]
8. com.sun.org.apache.xpath.internal.objects.XString[@ 431]
9. com.sun.org.apache.xpath.internal.objects.XStringForFSB[@ 425]
10. com.sun.security.auth.X500Principal[@ 137]
11. java.awt.RenderingHints[@ 648]
12. org.ietf.jgss.Oid[@ 144]
13. sun.tools.jconsole.inspector.XTree.Token[@ 599]

State comparison on same object

JDK 1.5

1. com.sun.org.apache.xerces.internal.impl.dv.xs.DoubleDV.XDouble[@ 125]
2. com.sun.org.apache.xerces.internal.impl.dv.xs.FloatDV.XFloat[@ 125]
3. java.net.SocketPermission[@ 824]

Equals overloading for similarity

JDK 1.5

1. com.sun.org.apache.xerces.internal.xni.XMLString
2. com.sun.org.apache.xpath.internal.objects.XStringForFSB
3. com.sun.tools.corba.se.idl.Token
4. java.awt.datatransfer.DataFlavor
5. sun.tools.jar.JarVerifierStream.CertCache
6. sun.tools.tree.BooleanExpression
7. sun.tools.tree.ConvertExpression
8. sun.tools.tree.DoubleExpression
9. sun.tools.tree.Expression
10. sun.tools.tree.FloatExpression
11. sun.tools.tree.IdentifierExpression
12. sun.tools.tree.IntegerExpression
13. sun.tools.tree.LongExpression
14. sun.tools.tree.NullExpression
15. sun.tools.tree.StringExpression

Tomcat 6.0

1. org.apache.jasper.xmlparser.XMLString
2. org.apache.tomcat.util.buf.ByteChunk
3. org.apache.tomcat.util.buf.CharChunk
4. org.apache.tomcat.util.buf.MessageBytes

Equals overloading without overriding

JDK 1.5

1. com.sun.corba.se.impl.io.ValueUtility.IdentityKeyValueStack.KeyValuePair
2. com.sun.jmx.snmp.agent.SnmpIndex
3. com.sun.org.apache.xerces.internal.impl.xs.identity.IdentityConstraint

4. com.sun.org.apache.xerces.internal.impl.xs.util.XInt
5. com.sun.org.apache.xerces.internal.xni.XMLString
6. com.sun.org.apache.xml.internal.dtm.ref.ExtendedType
7. com.sun.org.apache.xpath.internal.objects.XBoolean
8. com.sun.org.apache.xpath.internal.objects.XBooleanStatic
9. com.sun.org.apache.xpath.internal.objects.XNodeSet
10. com.sun.org.apache.xpath.internal.objects.XNull
11. com.sun.org.apache.xpath.internal.objects.XNumber
12. com.sun.org.apache.xpath.internal.objects.XObject
13. com.sun.org.apache.xpath.internal.objects.XRTreeFrag
14. com.sun.tools.corba.se.idl.Token
15. java.awt.geom.Area
16. sun.font.StandardGlyphVector.GlyphTransformInfo
17. sun.security.acl.AllPermissionsImpl
18. sun.security.jgss.krb5.Krb5NameElement
19. sun.tools.jar.JarVerifierStream.CertCache
20. sun.tools.tree.BooleanExpression
21. sun.tools.tree.ConvertExpression
22. sun.tools.tree.DoubleExpression
23. sun.tools.tree.Expression
24. sun.tools.tree.FloatExpression
25. sun.tools.tree.IdentifierExpression
26. sun.tools.tree.IntegerExpression
27. sun.tools.tree.LongExpression
28. sun.tools.tree.NullExpression

Tomcat 6.0

1. org.apache.jasper.xmlparser.XMLString
2. org.apache.tomcat.util.buf.ByteChunk
3. org.apache.tomcat.util.buf.CharChunk
4. org.apache.tomcat.util.buf.MessageBytes

This reference type test

JDK 1.5

1. com.sun.org.apache.xml.internal.utils.synthetic.reflection.EntryPoint[@ 169]

Appendix C

Reported Warnings

Note that false positives are marked with a *****. Also note that **@ num** means problematic pattern starts at **num** line number.

Use of non-equals method on field

JDK 1.5

1. com.sun.java.util.jar.pack.ConstantPool.ClassEntry[@ 325]
2. com.sun.java.util.jar.pack.ConstantPool.DescriptorEntry[@ 364]
3. com.sun.java.util.jar.pack.ConstantPool.MemberEntry[@ 423]
4. com.sun.java.util.jar.pack.ConstantPool.StringEntry[@ 293]
5. com.sun.org.apache.xerces.internal.impl.dtd.XMLDTDDescription[@ 155]
6. java.awt.datatransfer.DataFlavor[@ 886]
7. java.rmi.server.RemoteObject[@ 122]
8. sun.security.x509.GeneralName[@ 177]

Domain specific map implementation

JDK 1.5

1. com.sun.tools.jdi.LinkedHashMap[@ 711]
2. java.util.Collections.EmptyMap[@ 3025]

3. java.util.EnumMap[@ 607]
4. java.util.IdentityHashMap[@ 624]
5. javax.management.openmbean.TabularDataSupport[@ 628]

Wrapper implementation pattern

JDK 1.5

1. com.sun.management.GcInfo[@ 227]
2. java.util.Collections.CheckedList[@ 2413]
3. java.util.Collections.CheckedMap[@ 2567]
4. java.util.Collections.CheckedSet[@ 2310]
5. java.util.Collections.SynchronizedList[@ 1809]
6. java.util.Collections.SynchronizedMap[@ 2020]
7. java.util.Collections.SynchronizedSet[@ 1657]
8. java.util.Collections.UnmodifiableList[@ 1152]
9. java.util.Collections.UnmodifiableMap[@ 1320]
10. java.util.Collections.UnmodifiableSet[@ 1067]
11. java.util.jar.Attributes[@ 254]
12. javax.print.attribute.AttributeSetUtilities.SynchronizedAttributeSet[@ 328]
13. javax.print.attribute.AttributeSetUtilities.UnmodifiableAttributeSet[@ 117]
14. sun.management.LazyCompositeData[@ 42]

Tomcat 6.0

1. org.apache.catalina.tribes.tipis.AbstractReplicatedMap.MapEntry[@ 1317]
2. org.apache.jasper.el.JspMethodExpression[@ 82]
3. org.apache.jasper.el.JspValueExpression[@ 112]

Equals overloading with overriding

JDK 1.5

1. com.sun.corba.se.spi.ior.iiop.GIOPVersion
2. com.sun.java.util.jar.pack.Attribute.Layout
3. com.sun.java.util.jar.pack.Instruction
4. com.sun.java_cup.internal.action_part
5. com.sun.java_cup.internal.lalr_item
6. com.sun.java_cup.internal.lalr_item_set
7. com.sun.java_cup.internal.lalr_state
8. com.sun.java_cup.internal.lr_item_core
9. com.sun.java_cup.internal.nonassoc_action
10. com.sun.java_cup.internal.parse_action
11. com.sun.java_cup.internal.production
12. com.sun.java_cup.internal.production_part
13. com.sun.java_cup.internal.reduce_action
14. com.sun.java_cup.internal.shift_action
15. com.sun.java_cup.internal.symbol_part
16. com.sun.java_cup.internal.symbol_set
17. com.sun.java_cup.internal.terminal_set
18. com.sun.org.apache.xml.internal.dtm.ref.DTMNodeProxy
19. com.sun.org.apache.xml.internal.utils.XMLStringDefault
20. com.sun.org.apache.xpath.internal.objects.XString
21. com.sun.org.apache.xpath.internal.objects.XStringForFSB
22. java.awt.DisplayMode
23. java.awt.MenuShortcut
24. java.awt.datatransfer.DataFlavor
25. java.awt.font.FontRenderContext
26. java.awt.font.ImageGraphicAttribute

27. java.awt.font.ShapeGraphicAttribute
28. java.awt.font.TextHitInfo
29. java.awt.font.TextLayout
30. java.sql.Timestamp
31. sun.font.CoreMetrics
32. sun.font.StandardGlyphVector
33. sun.security.acl.GroupImpl
34. sun.security.jgss.GSSNameImpl
35. sun.security.jgss.ProviderList.PreferencesEntry
36. sun.security.util.BigInt
37. sun.security.util.DerInputBuffer
38. sun.security.util.DerValue
39. sun.security.util.ObjectIdentifier
40. sun.security.x509.AlgorithmId
41. sun.security.x509.X509Cert
42. sun.security.x509.X509CertInfo
43. sun.tools.tree.StringExpression

Lucene 3.0

1. org.apache.lucene.spatial.geometry.shape.Vector2D

Tomcat 6.0

1. org.apache.el.ValueExpressionLiteral

Path generation reaching cut-off

JDK 1.5

1. com.sun.jndi.dns.DnsName
2. com.sun.org.apache.xerces.internal.jaxp.datatype.DurationImpl

3. com.sun.org.apache.xerces.internal.jaxp.datatype.XMLGregorianCalendarImpl
4. com.sun.org.apache.xerces.internal.util.URI
5. com.sun.org.apache.xml.internal.utils.URI
6. com.sun.security.auth.SubjectCodeSource
7. java.awt.Font
8. java.net.URI
9. java.security.UnresolvedPermission
10. java.text.DateFormatSymbols
11. javax.management.openmbean.OpenMBeanInfoSupport

Appendix D

Supportable Unknown Code Patterns

Use of a field to represent an array length

JDK 1.5

1. com.sun.jmx.snmp.SnmpOid
2. java.awt.image.ColorModel
3. java.awt.image.ComponentColorModel
4. java.awt.image.PackedColorModel
5. java.lang.String
6. java.net.Inet6Address
7. java.text.DigitList
8. sun.awt.robot.ScreenCapture
9. sun.text.CompactByteArray
10. sun.text.CompactCharArray
11. sun.text.CompactIntArray
12. sun.text.CompactShortArray
13. sun.security.util.BitArray

Lucene 3.0

1. org.apache.lucene.analysis.Token
2. org.apache.lucene.index.Payload

Use of an array as a set

JDK 1.5

1. java.security.AccessControlContext
2. java.security.CodeSource
3. sun.security.provider.SelfPermission

Multi-Dimensional array equality pattern

JDK 1.5

1. com.sun.corba.se.impl.ior.StubIORImpl

Collection operations on field

JDK 1.5

1. com.sun.corba.se.impl.ior.ObjectAdapterIdArray
2. com.sun.jndi.ldap.LdapName
3. com.sun.jndi.ldap.LdapName.Rdn
4. java.awt.datatransfer.MimeTypeParameterList
5. java.net.NetworkInterface
6. javax.management.openmbean.CompositeDataSupport
7. javax.naming.NameImpl
8. javax.naming.directory.BasicAttributes
9. javax.naming.ldap.LdapName
10. javax.naming.ldap.Rdn
11. javax.print.attribute.HashAttributeSet

12. javax.swing.text.StyleContext.SmallAttributeSet
13. sun.security.pkcs.PKCS10Attributes

Lucene 3.0

1. org.apache.lucene.search.MultiPhraseQuery
2. org.apache.lucene.search.spans.SpanOrQuery

Handling data flow of boolean type

JDK 1.5

1. java.beans.IndexedPropertyDescriptor
2. java.beans.PropertyDescriptor
3. java.net.InetSocketAddress
4. java.rmi.dgc.VMID
5. java.security.Identity
6. sun.rmi.transport.tcp.TCPEndpoint
7. sun.util.calendar.CalendarDate

Control dependency

JDK 1.5

1. com.sun.jmx.snmp.IPAcl.NetMaskImpl
2. javax.security.auth.PrivateCredentialPermission
3. sun.security.jgss.GSSCredentialImpl

Appendix E

Unsupportable Unknown Code Patterns

Domain specic representation on array

JDK 1.5

1. com.sun.jlex.internal.SparseBitSet
2. java.text.ChoiceFormat
3. java.text.MessageFormat
4. java.util.BitSet
5. com.sun.jmx.snmp.IPAcl.PrincipalImpl
6. javax.swing.text.TabSet

Lucene 3.0

1. org.apache.lucene.util.OpenBitSet

Domain specic representation on field

JDK 1.5

1. com.sun.corba.se.impl.naming.pcosnaming.InternalBindingKey
2. com.sun.corba.se.impl.naming.cosnaming.InternalBindingKey

3. com.sun.org.apache.xerces.internal.impl.dtd.models.CMStateSet
4. com.sun.org.apache.xerces.internal.impl.dv.xs.DecimalDV.XDecimal
5. com.sun.org.apache.xerces.internal.impl.xs.traversers.XSDHandler.XSDKey
6. java.util.EnumMap.EntryIterator
7. java.util.IdentityHashMap.EntryIterator
8. java.util.SimpleTimeZone
9. javax.naming.directory.BasicAttribute

Lucene 3.0

1. org.apache.lucene.util.AttributeSource

Comparison delegated to a field

JDK 1.5

1. com.sun.org.apache.xerces.internal.impl.dv.xs.
   AbstractDateTimeDV.DateTimeData
2. com.sun.tools.apt.mirror.type.TypeMirrorImpl
3. com.sun.tools.javac.code.Types.SingletonType
4. com.sun.tools.javac.code.Types.TypePair
5. java.io.File
6. java.net.URL
7. java.util.jar.Attributes.Name
8. javax.rmi.CORBA.Stub
9. org.omg.CORBA.portable.ObjectImpl

Polymorphic field

JDK 1.5

1. com.sun.jndi.ldap.DigestClientId
2. com.sun.jndi.ldap.SimpleClientId

Creation of new state for equality

JDK 1.5

1. javax.rmi.ssl.SslRMIServerSocketFactory
2. javax.security.auth.Subject
3. sun.reflect.generics.reflectiveObjects.ParameterizedTypeImpl
4. sun.security.provider.certpath.CertId
5. sun.security.x509.CRLExtensions
6. sun.security.x509.CertificateExtensions
7. sun.security.x509.IPAddressName
8. sun.security.x509.KeyIdentifier
9. sun.security.x509.OtherName
10. java.util.GregorianCalendar
11. sun.util.BuddhistCalendar

Wrapped state Comparison

JDK 1.5

1. java.util.Collections.CheckedMap.CheckedEntrySet.CheckedEntry
2. java.util.Collections.UnmodifiableMap.UnmodifiableEntrySet.
   UnmodifiableEntry

Domain specic equality

JDK 1.5

1. java.security.spec.ECPoint