An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java

Chandan R. Rupakheti, Daqing Hou
Department of Electrical and Computer Engineering
Clarkson University, Potsdam, NY 13699
{rupakhr, dhou}@clarkson.edu

Abstract—The standard collection data structures in Object-Oriented languages require their element classes provide a predicate to compare two objects for equality. Among other correctness requirements, this predicate must be an equivalence relation. The chance of mistakes increases when equality is implemented in a type hierarchy. Detection of such problems requires reasoning about the equality at a higher level semantics than code, involving the state, the behavior, and the sub-typing relations in the type hierarchy. Presenting a path-based, abstraction-oriented approach to checking the correctness of equality implementation in a type hierarchy. In our approach, code patterns on paths are identified and translated into abstractions in Alloy. The Alloy model is then automatically checked to reveal any problems related to the equivalence relation. Our evaluation shows that this approach (1) found problems in production code, and (2) scaled to a project as large as JDK 1.5. We believe that it has the potential to be used on a developer’s desktop on a daily basis.

I. INTRODUCTION

Object-oriented programming languages such as Java and C# provide a useful set of collection data types such as lists, sets, and maps. In order to work as elements inside these containers, application objects need to support an equality predicate with which a pair of objects can be compared. In Java, as its behavioral specification, the equals method is required to honor the reflexivity, symmetry, and transitivity properties of the equivalence relation.

The issue of object equality is a general, long-standing design problem that can be traced back to Lisp [2]. Its impact can be wide. For example, 623 classes in JDK 1.5 implement an equals method, covering areas such as security, corba implementation, utility classes, collection types, and GUI.

The current practice with object equality still has ample room for improvements. To respect the equality contract, several considerations must be taken, which can be easily overlooked by a developer, resulting in buggy or fragile code. Such defects can be notoriously difficult to find, even for an experienced programmer [5] [21] [17] [15]. Our current understanding of the problem is so vague that the wizards in popular Java IDEs such as Eclipse 3.5 [2] and NetBeans 6.8 [3] cover only a subset of the ways in which equals can be implemented. Even textbooks sometimes contain incorrect implementations of equals (e.g., the Point3D class in [13], pp. 182 violates the transitivity property). Finally, to our best knowledge, existing static checkers for Java do very little in terms of detecting violations of the equivalence properties (Section VI-D).

The persistence of equality related problems motivated us to design our own checker. The key idea is to use the checker to look for high level abstractions in the source code to formulate a logical model, which can subsequently be model checked. There is a set of code patterns/abstractions commonly used in defining and implementing equals. For instance, the equality of two points can be defined in terms of their coordinates x and y, which is implemented in the equals method of the Point class shown in Listing 1.

Our checker requires inter-procedural paths for program analysis [4]. We use the intra-procedural control flow graphs of Soot (Jimple) [20] to produce inter-procedural paths. In the process, only methods deemed necessary are expanded (Section III). On top of the inter-procedural paths, we run several code pattern detection algorithms to automatically identify common abstractions used in defining equality (Section IV). In doing so, we exploit both the structure and the behavior (temporal order) of the path nodes. The detected abstractions are then translated into an Alloy model [11]. The Alloy analyzer then model checks and reports any property violation in the equals implementations. Our checker produces one Alloy model for every type hierarchy.

Besides model checking, our checker also detects other problems such as NullPointerException and ClassCastException using its path-sensitive data flow analysis framework. However, this paper focuses mainly on the overall design and implementation of the checker and its use in the identification of problems related to the equivalence relationship.

This work makes two contributions:

1) A two-layered approach. We perform program analysis in two layers. The first layer is the low-level error detection through a path-sensitive, data-flow analysis built on top of the intermediate representation of Soot (Jimple). The second layer translates these low-level programming details into a high-level logical model in Alloy and model-checks it for constraint violations.

1http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html. All URLs verified on June 23, 2010.
2http://www.eclipse.org/eclipse/
3http://www.netbeans.org/
2) The implementation and evaluation of a static checker for the approach. Based on the results of evaluating the checker reported in Section V, we conclude that the checker can scale up to large projects and be useful.

A. A Motivating Example

To get an overview of the whole process, consider the two classes Point and PolarPoint shown in Listings 1 and 2. We assume that Point and PolarPoint form a behavioral subtyping relationship [14]. Furthermore, PolarPoint extends Point to provide the r and \( \theta \) coordinates (Listing 2) in addition to \( x \) and \( y \), and reuses the internal representations of Point.

Listing 1: A Point class in cartesian coordinate.

```java
public class Point {
    protected int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    // ... getx() and gety() defined here
    public boolean equals(Object o) {
        if (o == null) return false;
        if (o.getClass() != Point.class) return false;
        Point that = (Point)o;
        return this.x == that.x & this.y == that.y;
    }
}
```

Listing 2: A PolarPoint class in polar coordinate.

```java
public class PolarPoint extends Point {
    public PolarPoint(int x, int y) {super(x, y);}    // super
    public double getTheta() {...
    public double getR() {...
}
```

Given the type hierarchy containing these two classes (Point and PolarPoint), our checker produces the Alloy model shown in Listing 3. Taken this model as input, the Alloy analyzer reports a violation of the reflexivity property with a counterexample shown in Figure 1. The reason is that the type checking statement in line 10 (Listing 1) prevents any two PolarPoint’s from being equal. The following Java code illustrates the nature of the counterexample:

```
PolarPoint a = new PolarPoint(7,7);
a.equals(a); // returns false
```

To see the counter-intuitive effect of such a violation upon the data structures of the Java Collection Framework, consider the following code snippet that uses a `java.util.ArrayList`

```
ArrayList<PolarPoint> list = ...;
PolarPoint a = new PolarPoint(7,7);
list.add(a); // adds a to the list
list.contains(a); // returns false
```

The `contains()` method of the `ArrayList` calls the `equals()` method of `PolarPoint` to check for the containment. Since the `equals()` method violates reflexivity, the list cannot find the point added before, and thus, behave incorrectly by returning `false`. This is a key reason why the correctness of `equals` is so important in Java.

```java
Listing 3: An Alloy model for the Point type hierarchy.

abstract sig Object {} // Signature Decl.
sig P o i n t e x t e n d s O b j e c t {
    Point_y : Int, // Field Declaration
    Point_x : Int
}
sig PolarPoint extends Point {} // Equals Predicate
pred Object ::= equals(that: Object) {
    (this in Point - PolarPoint) => // For Point
        (that in Point - PolarPoint) and
        (this.Point_x = that.Point_x) and
        (this.Point_y = that.Point_y)
    } else
        (this in PolarPoint) => // For PolarPoint
        (that in Point - PolarPoint) and
        (this.Point_x = that.Point_x) and
        (this.Point_y = that.Point_y)
} // Equivalence relation specification
assert reflexive {all a : Object | a.equals[a]}
assert symmetric {all a, b : Object | a.equals[b] <=> b.equals[a]}
assert transitive {all a, b, c: Object | a.equals[b] and b.equals[c] => a.equals[c]}
```

II. Checker Architecture

Our checker implements an inter-procedural, path-based analysis as an Eclipse plugin. It is built on top of Eclipse’s Java Development Tool (JDT), Soot [20], and Alloy [11].

The top-level architecture of the checker is shown in Figure 2. It takes a Java project as input. First, JDT is used to search for all classes that implement an `equals`. Our checker constructs type hierarchies that contain any of these classes. It then loads one type hierarchy at a time using Soot to generate inter-procedural paths. More specifically, it does so for all the `equals` that are either directly implemented or inherited by the concrete classes in the type hierarchy.

The Jimple-based, intra-procedural control flow graph from Soot [20] is used to produce the inter-procedural paths. Specif-

![Fig. 1: A counter example for a reflexivity violation in the equality of the Point type hierarchy.](image-url)
A. FlowGraph

The FlowGraph function called at lines 1 and 18 of the main algorithm (Algorithm 1) returns a control flow graph for a given method. This is a directed flow-graph constructed from the Jimple-body of the method being processed. The flow graphs for equals and eq are shown in Figures 3a and 3b, respectively. Note that we are presenting the flow graphs based on the Java code and not the Jimple code for illustration and simplicity. The actual graphs from Soot contain more nodes than these due to the lower level Jimple representation used.

Listing 4: An equals implementation for Person class.

```
public class Person implements Cloneable {
    private char[] name;
    private char[] address;
    public boolean equals(Object o) {
        if (o == null || this.getClass() != o.getClass())
            return false;
        boolean t = this.eq(this.name, that.name);
        if (t != true)
            return false;
        t = that.eq(that.address, this.address);
        if (t != true)
            return false;
        return true;
    }

    protected boolean eq(char[] a, char[] b) {
        if (a.length != b.length)
            return false;
        int i = 0;
        while (i < a.length) {
            if (a[i] != b[i])
                return false;
        }
        return true;
    }
}
```

B. IntraProcedurePathGenerator

The IntraProcedurePathGenerator algorithm (Algorithm 2) takes a flow graph $G = (V, E)$ as input and returns a set of intra-procedural paths $S$ as output. It assumes that the graph must have one source, at least one sink, and may contain cycles. IntraProcedurePathGenerator is called at lines 2 and 19.
of the main algorithm. For the flow graphs in Figures 3a and 3b, this algorithm produces the path enumerations shown in Listings 5 and 6, respectively.

Note that in our algorithm, a loop body is entered at most once on a path. Generally, for each loop, two paths are extracted: one for the case when the loop condition is false (e.g., path 2 of Listing 6), and the other when the loop condition is true (e.g., path 4 of Listing 6). In the latter case, the loop condition will appear again on the same path with a false value. We call this approach 1-2 loop unrolling.

Listing 5: Intra-procedural paths for Person.equals().

Listing 6: Intra-procedural paths for Person.eq().

Although the worst-case running time for this algorithm is $O(|E||V|)$, in practice, it works sufficiently well for the equals method for the majority of the cases (Section V).

C. Context

Our notion of a calling context captures the full call stack starting from the entry method equals. This is done by associating a context object with each intra-procedural path. The context object consists of three entities: a call site, a resolved method, and a receiver class. Specifically, the call site is the path node from which the current path is expanded, the resolved method represents the method from which this path originates, and the receiver class is the runtime class for the resolved method.

Because the entry method equals() has no call site, null value is used in its context object. This is why the first call to Context function at line 4 of the main algorithm assigns null as a call site to the paths. Therefore, all of the paths in Listing 5 are assigned a context object $C_1 = [null, Person.equals(), Person]$.

The need for the receiver class becomes apparent in the following situation. Suppose the Person class is inherited by another class Developer that overrides only the eq method.
this case, we also want to analyze the equals method of the Person class with Developer as a receiver class. The context object is null, Person.equals(), Developer].

The call to Context at line 20 of the main algorithm assigns a context object to each new path generated for intermediate method calls other than equals. For example, Listing 4 contains two such method calls (to Person.eq) at lines 8 and 11. To create inter-procedural paths, the intra-procedural paths for Person.eq in Listing 6 are copied twice: one for the this.eq() call at line 8 and the other for the that.eq() call at line 11. Each path copied for this.eq() is assigned a context object $C_2 = [l_8, Person.eq(), Person]$, and a context object $C_3 = [l_{11}, Person.eq(), Person]$ for that.eq(). $l_8$ and $l_{11}$ represent the two call sites at lines 8 and 11 of Listing 4.

Because each path node can access its context object, a full call stack can be constructed dynamically. Consider node 25 as an example. Since node 25 has different context objects on the paths expanded for call site nodes 8 and 11, let us assume that we are processing a path that has the context object $C_3$. Thus, its call site is node 11. The context object of node 11, in turn, is $C_1$, whose call site is null. This traversal of context objects continues until a null call site node is encountered, and all the call sites encountered during the traversal form the call stack. This results in a full context sensitivity.

D. Filter

The Filter function called at lines 5 and 24 removes infeasible paths from the input path set. It applies several path-sensitive data flow analyses to the paths to compute facts at path nodes, which are used for constraints checking and path pruning. Four filters are developed:

- **Boolean Value Filter:** This is a boolean value copy propagation analysis run over the generated paths of the equals method. It eliminates infeasible paths that contain contradictions in terms of values of boolean variables (examples of this can be found in Section III-G). Since equality is a boolean predicate and it is sufficient to model the equality with true-returning paths, this filter also prunes paths that return false. For example, it prunes all of the paths in Listing 5 except path 4 (marked with a *). So, at line 7, the main algorithm adds only path 4 to the queue for further processing.

- **Nullness Constant Filter:** It performs copy propagation of null constant for reference variables. Like BooleanFilter, it eliminates infeasible paths related to null.

- **Type Analysis Filter:** This filter accumulates type information for variables. For example, if the type comparison at line 5 of Listing 4 returns true, the filter can infer that the type of $o$ is Person. Such type information, combined with CHA, is used to filter infeasible path as well as for increasing the precision of the receiver types.

- **Throw Analysis Filter:** This filter prunes a path that throws an exception as we are only interested in true-returning paths.

E. Expandable

This function, called at line 15 of the main algorithm, makes a decision about method expansion. The expandable function detects several pre-defined patterns in the code involving method invocation to determine whether a method call can be directly recognized as a relevant abstraction or not. If the function determines that an abstraction is not possible then it returns true, indicating that method expansion is necessary. The use of abstraction whenever possible saves both time and space. Furthermore, this sometimes also makes the otherwise infeasible pattern detection feasible by hiding the details of underlying implementations.

The rules for abstracting method calls are as follows:

- **Getters:** Getter methods involved in comparison such as this.getState() == that.getState() are abstracted directly as this.state = that.state in Alloy without expansion. This also includes the case of this.getState().equals(that.getState()).

- **Static methods:** Static method involved in comparison such as StaticMethod(this.f) == StaticMethod(that.f) is abstracted as this.StaticMethod_f = that.StaticMethod_f in Alloy without expansion. This also includes the case of StaticMethod(this.f).equals(StaticMethod(that.f)).

- **this.f.m():** To keep the implementation simple, a method call on a field is not expanded. This sometimes helped detect abstractions that would otherwise go undetected due to the implementation details inside m(). We found only 5 cases in JDK 1.5 and none in Tomcat 6 and Lucene 3.0 that resulted in undetected code patterns because of this heuristics.

- **Collection methods:** Methods of List, Set, Map, and Collection interfaces from the java.util package, such as Collection.iterator() and List.size(), are not expanded as they have predefined meaning.

- **Object’s methods:** Methods such as getClass() and hashCode() are not expanded as they have predefined meaning. For example, the getClass() calls at node 5, path 4 in Listing 5 (line 5 of Listing 4) are not expanded.

F. CHA

Generally, type resolution using CHA results in the type and all of the subtypes to be included in the target type set for the receiver object of a virtual method. Our CHA function also performs data flow analysis on a path to accumulate the type information to reduce the number of possible target types. For instance, suppose that the Person class has a subtype, Developer, that overrides eq. Consider path 4 in Listing 5, which passes the filter at line 5 of the main algorithm and is later dequed at line 10. The CallSite function at line 11 of the main algorithm returns node 8 first, whose receiver object is this. After processing this call site and updating the path successively, it returns node 11, whose receiver object is that. The Expandable function at line 15 cannot abstract these calls, 6In equals(), exceptions are almost always used as defensive programming without direct contribution to equality, and rarely propagated across methods.

6Recursive calls are expanded only once.

8Instead of simply returning a field, a getter may compute an abstract state in a way that if expanded, the abstraction detectors may fail to recognize.
so the CHA function at line 16 is called. For this object, there is only one target \{Person.eq()\}, but for that object, the regular CHA would return two targets \{Person.eq(), Developer.eq()\}. However, due to the type information in node 5 accumulated through the data flow analysis, our CHA precisely returns \{Person.eq()\}.

G. Inline

Intra-procedural paths from callees are inlined into a caller’s paths to form inter-procedural paths. Consider the this.eq() call site (node 8) appearing on path 4 in Listing 5 and the five intra-procedural paths for Person.eq() shown in Listing 6. Path 4 is cloned and the five intra-procedural paths are inlined into the cloned path (the Inline function at line 23 of the main algorithm). The resulting paths after inlining are shown in Listing 7. When a path is cloned for inlining, the call site to be expanded is also cloned in the new path; the two call sites represent the method entry and return points, sandwiching the inlined path nodes (e.g., node 8 appears twice in each path to represent entry and return points). Among all of these paths, only paths 2 and 4 (marked with *) pass the BooleanFilter called at line 24 of the main algorithm.

Listing 7: Intermediate paths for Person.equals() after the eq at line 8 inlined.

As an example of an infeasible path, consider path 1 (Listing 7) and trace the data flow in Listing 4. The data flow analysis on path 1 reveals that the value of variable \( t \) at the end of line 8 (represented by the second 8 in path 1) is false. This means that the condition at line 9 will be true and, thus, control should branch to line 10. This contradicts with path 1, which indicates that control will go to line 11 instead. Therefore, path 1 is infeasible and can be pruned.

Listing 8: Final inter-procedural paths for Person.equals().

The mapping between paths and Alloy can be one-to-one or many-to-one. For example, the Alloy model for the Point type hierarchy shown in Listing 3 performs one-to-one translation of all comparison statements in the Java code to the Alloy facts. On the other hand, a Java class that uses composite data types (like an array) has several comparison statements scattered over multiple paths to produce the equality logic. For instance, the comparison logic of name and address fields of the Person class (Listing 4) is scattered over 4 paths (Listing 8). However, its Alloy model shown in Listing 9 is produced through the many-to-one translation where the array equality pattern detector (Section IV-F) abstracts the scattered logic in all of the paths to one comparison of sequence of integer (seq Int) for each array field in the model.

A. An Example of Alloy Model

The example Alloy model in Listing 9 is for Person (Listing 4). We assume that a Developer class extends Person without overriding anything related to equals. Note that for space reason, we have modified the generated model. Particularly, qualified names are not used.

Line 1 in the listing declares the java.lang.Object class as a type Object in Alloy using the signature construct. The subtyping relation in Java is straightforwardly modelled as Alloy’s subtyping. In line 3, java.lang.Cloneable interface is declared as an Alloy type. The facts that Person extends Object and implements Cloneable interface in Java are modeled at lines 5 and 9 in Alloy. In general, because the extends construct in Alloy permits only single inheritance, multiple inheritance such as this one can be modeled using both the extends and in keywords as done in the fact statement at line 9 [11].

The Developer class is declared at line 11. The character array fields are abstracted as sequence of integer at lines 6 and 7. In general, we model only array, set, map, and list types in Alloy. All the other data types for fields are modeled as integer.

The equals method in Java is represented as an Alloy predicate Object::equals that takes Object as a parameter (line 14). Structurally, the body of the equals predicate consists of multiple what we call the implication blocks, separated by the else keyword. The equals method for each concrete class in a type hierarchy results in one implication block. For example, the implication blocks for Person and Developer
range from lines 15 to 20, and lines 22 to 27, respectively. Each implication block, in turn, consists of one or more fact blocks, separated by an or keyword, each of which represents a different way of defining equality. However, in this example, there is only one fact block in each implication block.

Listing 9: Alloy model for the Person type hierarchy.

```alloy
t abstract sig Object {}
sig Cloneable in Object {}
sig Person extends Object {
  name : seq Int ,
  address : seq Int
}
fact { Person in Cloneable }
sig Developer extends Person {}

// Equals predicate
pred Object :: equals( that : Object ) {
  (this in Person − Developer) => // Person
  (that in Person − Developer) and
  (this . name = that . name) and
  (this . address = that . address)
}
else
  (this in Developer) => // Developer
  (that in Developer) and
  (this . name = that . name) and
  (this . address = that . address)
}

```

The runtime type checking this.getClass() == that.getClass() results in the two facts at lines 17 and 24, for Person and Developer, respectively. Even though the Developer class inherits the equals method from Person, our model generation logic ensures that dynamic dispatching and runtime type checking information is properly handled to produce the correct Alloy model, and that is why lines 17 and 24 are different.

For this particular model, the Alloy analyzer does not report any violation.

B. Copy Root Analysis

The copy root of a variable at a given point on a path is the expression from which this variable gets its value. It is evaluated recursively with the following rules:

1) \( \text{CopyRoot(Var)} = \text{Expression} \): This rule applies if the path has a definition statement \( \text{Var} = \text{Expression} \) that is not killed before the evaluation point. For example, for \( [x = a-b; \text{return } x] \), \( \text{CopyRoot}(x) = a-b \) after \( x = a-b \).

2) \( \text{CopyRoot(Var)} = \text{CopyRoot(Var1)} \): If the path has a definition statement \( \text{Var} = \text{Var1} \) that is not killed before the evaluation point. For example, for \( [x = \text{this}; y = x; \text{return } y] \), \( \text{CopyRoot}(y) = \text{this} \) after \( y = x \).

3) \( \text{CopyRoot(Formal Parameter)} = \text{Formal Parameter} \) for entry method.

4) \( \text{CopyRoot(Formal Parameter)} = \text{CopyRoot(Actual Parameter)} \): On paths, we bind actual parameters with formal parameters for non-entry method.

C. Object Root Analysis

The object root of a variable at a given point on a path is the root of an object tree, from which, or any of its components, this variable gets its value. It is evaluated recursively with the following rules:

1) \( \text{ObjectRoot(this)} = \text{this} \): Object root of this is itself.

2) \( \text{ObjectRoot(r.f)} = \text{ObjectRoot(r)} \): e.g., ObjectRoot(this.field) = ObjectRoot(this) = this.

3) \( \text{ObjectRoot(a[e])} = \text{ObjectRoot(a)} \): The object root of an array access is the object root of the array.

4) \( \text{ObjectRoot(Var)} = \text{ObjectRoot(Expression)} \): This rule applies if the path has a definition statement \( \text{Var} = \text{Expression} \) that is not killed before the evaluation point. For example, for path \([a = \text{this}; b = a; c = b.field; d = c; \text{return } d] \), after \( d = c \), it holds \( \text{ObjectRoot(d)} = \text{ObjectRoot(c)} = \text{ObjectRoot(b.field)} = \text{ObjectRoot(b)} = \text{ObjectRoot(a)} = \text{ObjectRoot(this)} = \text{this} \).

5) \( \text{ObjectRoot(Formal Parameter)} = \text{Formal Parameter} \) for entry method.

6) \( \text{ObjectRoot(Formal Parameter)} = \text{ObjectRoot(Actual Parameter)} \) for non entry method.

7) \( \text{ObjectRoot(Others)} = \text{null} \): Object root of any other expression not listed above, such as static field, return value of a static method, or a constant, is null.

Listing 10: An equals implementation for Circle.

```alloy
boolean equals(Object o) {
  boolean t = o instanceof Circle;
  if (t != true)
    return false;
  Circle that = (Circle)o;
  int thisR = this.radius;
  int thatR = that.radius;
  if (thisR != thatR)
    return false;
  return true;
}
```

D. Type Checking Pattern

To illustrate how \( \text{CopyRoot} \) is used in detecting type checking patterns, consider the \( \text{equals} \) method for the \( \text{Circle} \) class shown in Listing 10, which is written in a way to closely resemble the Jimple code that is actually used for analysis. The true-returning path for the \( \text{equals} \) is \([1,2,3,5,6,7,8,10]\). The pattern detector iterates through the nodes in the path looking for a comparison. It finds one at line 3. Then it checks if \( \text{CopyRoot(t)} \) is an \( \text{instanceof} \) expression or not, which is true. So it further checks if an operand of the \( \text{instanceof} \) expression is the parameter of \( \text{equals} \). This is also true \( \text{(CopyRoot(o)} == o) \). Thus, it concludes that line 3 performs an \( \text{instanceof} \) type checking and abstracts it as \( \text{that in Circle} \) in Alloy.

In general, the pattern detectors are programmed to detect the following type checking patterns:
• o instanceof Type: This predicate returns true if o is of Type itself or any of its subtypes.
• o.getClass() == this.getClass(): o has the same type as this.
• o.getClass() == Type.class: o is of Type.
• try {Type that = (Type); . . .} catch(ClassCastException e) { return false;}: An exception is thrown if o is not of Type or any of its subtypes.

E. State Comparison Patterns

To introduce state comparison patterns and how they are translated into Alloy, let us reconsider the true-returning path \[1,2,3,5,6,7,8,10\] in Listing 10. The pattern detector iterates through each node in the path looking for

• a comparison of two local variables. Note that in Jimple, comparisons are always between local variables or constants. It finds one at line 8.
• verifying that CopyRoot(thisR) and CopyRoot(thatR) are field references, which is true in this case.
• verifying that the following holds: ObjectRoot(thisR) = this & ObjectRoot(thatR) = o, or ObjectRoot(thisR) = o & ObjectRoot(thatR) = this.

In this case, all conditions are satisfied and, thus, the detector concludes that line 8 is comparing two fields and abstracts it as this.radius=that.radius in Alloy.

In general, the pattern detectors are programmed to detect following state checking patterns, where f represents a field, m() a virtual method, SM() a static method, and \[\{f\},\{m()\}\] the chaining of one or more fields or methods:

1) \[this[\{f\},\{m()\}]^+ = \theta that[\{f\},\{m()\}]^+\]

or

\[this[\{f\},\{m()\}]^+.equals(\theta that[\{f\},\{m()\}]^+)\].

In this case, instead of modeling a field type or expanding a method called on a field, we introduce new fields in Alloy. For example, this.f.m() == that.getG() is translated to this.f.m() = that.g, and this.f.m().equals(that.f) into this.f.m = that.f in Alloy. Here, we introduce fields f.m and in general, translate a getter such as getG() into g in the Alloy model.

2) \[this[\{f\},\{m()\}]^+.compareTo(\theta that[\{f\},\{m()\}]^+) = 0\]

is translated into a comparison of two fields in Alloy. For example, this.f1.compareTo(that.f2) == 0 is translated to this.f1 == that.f2.

3) \[SM(this[\{f\},\{m()\}]^+) = \theta SM(that[\{f\},\{m()\}]^+)\].

For example, Math.atan(this.f) == Math.atan(that.f) is translated to this.Math.atan.f = that.Math.atan.f in Alloy.

4) \[f[m()] \oplus \theta f[m()]\]

is translated to this.AND_f.f = that.f.

F. Equality Patterns Involving Composite Data Structures

Our checker can handle code patterns that involve four composite data structures and translate them to Alloy: arrays, lists, sets, and maps. Specifically, arrays and lists (subtypes of java.util.List) are represented as a sequence of integer in Alloy \((seq Int)\), sets (subtypes of java.util.Set) as set of integer \((set Int)\), and maps (subtypes of java.util.Map) as an integer to integer mapping \((Int \rightarrow Int)\).

Now let us look at how such code patterns are detected and translated to Alloy using arrays as an example. For this, consider the Person.eq() method in Listing 4 and the corresponding paths in Listing 6. Among these paths, only the two marked with * are inlined to the paths generated for Person.equals() method. In path 2, the length of two arrays are first compared at node 17, and control goes to the array bound check at node 20, which is false and, thus, the control returns true at node 25. For path 4, we have the array length check and bound check similar to path 2, but the bound check condition at node 20 is true and, thus, control goes to the array element comparison at node 21. The control at node 21 is false and control again goes back to the bound check at node 20 and then to node 25.

In the set of paths for a method, if any two exhibit these conditions, we conclude that two arrays are being compared in the method. We then abstract them as this.array = that.array in Alloy, as well as adding the array field \((array: seq Int)\) to the sig for the class under analysis.

The checker can also handle implementations of equals in subclasses implementing the three collection interfaces java.util.List, java.util.Set, and java.util.Map, including the standard implementations of these data structures inside JDK.

The logics of handling these implementations rely on the specified semantics of the standard interface methods and are robust in the presence of syntactic variations. They are similar to how array comparisons are handled and, thus, are skipped for the sake of brevity.

V. Evaluation Results

<table>
<thead>
<tr>
<th>Description</th>
<th>Tomcat 6</th>
<th>Lucene 3.0</th>
<th>JDK 1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces</td>
<td>175</td>
<td>66</td>
<td>1745</td>
</tr>
<tr>
<td>Classes</td>
<td>1237</td>
<td>888</td>
<td>11240</td>
</tr>
<tr>
<td>Type hierarchies</td>
<td>28</td>
<td>67</td>
<td>484</td>
</tr>
<tr>
<td>Defined equals</td>
<td>29</td>
<td>96</td>
<td>623</td>
</tr>
<tr>
<td>Processed equals</td>
<td>31</td>
<td>98</td>
<td>694</td>
</tr>
</tbody>
</table>

TABLE I: Characteristics of the inspected projects.

The equals checker has been tested with three medium to large, open source, real world projects: Tomcat 6 \(^{10}\), Lucene 3.0 \(^{11}\), and JDK 1.5. It successfully identified many code patterns and more importantly, discovered several problems related to the equivalence relation. Table I summarizes these projects in terms of the number of classes, interfaces and equals implementations. Tomcat 6, the smallest of the three, is a web server for hosting servlets; Lucene is a search engine library; and JDK 1.5 is the largest of the three. Note that sometimes, equals of a super class calls methods overridden in subclasses. In such scenarios, equals from the super has to be processed in the context of subclasses. That is why the values in the processed equals row in Table I are larger than those in the defined equals row.

\(^{10}\)http://tomcat.apache.org/

\(^{11}\)http://lucene.apache.org/
Table II: Summary of path generation and filtering.

<table>
<thead>
<tr>
<th>Description</th>
<th>Total paths</th>
<th>Tomcat 6</th>
<th>Lucene 3.0</th>
<th>JDK 1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean path filtering</td>
<td>515</td>
<td>727</td>
<td>27925</td>
<td></td>
</tr>
<tr>
<td>Nullness analysis filtering</td>
<td>0</td>
<td>0</td>
<td>845</td>
<td></td>
</tr>
<tr>
<td>Type analysis filtering</td>
<td>0</td>
<td>0</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Throw analysis filtering</td>
<td>0</td>
<td>0</td>
<td>3043</td>
<td></td>
</tr>
<tr>
<td>Other intermediate paths</td>
<td>45</td>
<td>132</td>
<td>8259</td>
<td></td>
</tr>
<tr>
<td>Final working paths</td>
<td>77</td>
<td>321</td>
<td>4591</td>
<td></td>
</tr>
<tr>
<td>Path reduction</td>
<td>85.04%</td>
<td>72.79%</td>
<td>89.74%</td>
<td></td>
</tr>
</tbody>
</table>

Table III: Summary of detected abstractions.

A. Path Generation and Filtering

Table II summarizes path generation for the three projects. The total paths constitute the filtered paths, the intermediate paths, and the final paths. The main problem with a path based analysis has been considered to be the path explosion. However, the data in the table show that our path reduction strategies reduce paths ranging from 72.79% to 89.74%.

Note that we have set a threshold of 512 paths per equals method. Experimentally, this limit works for the majority of the cases. Only 11 equals in JDK surpass this threshold, and none in Tomcat and Lucene.

B. Detected Abstractions

Table III summarizes the six abstractions our checker currently detects. While analyzing the projects, we came across some unanticipated comparison patterns that could not be handled by the current implementation of the available detectors. Upon a manual analysis of these equals implementation we found that about 40 of the 80 cases in JDK 1.5 and all 6 cases in Lucene 3.0 can be handled after some additional but straightforward extension to our checker. The remaining cases need further research. Due to space limitation, we will not provide further details in this paper.

C. Detected Problems

Table IV summarizes the problems related to the equivalence relation, where the numbers represent faulty type hierarchies. We want to emphasize that to reason about each faulty hierarchy, a developer must inspect and reason about a potentially much larger number of classes. For example, the 21 violations for JDK 1.5 affect more than 100 classes. For each detected violation, the checker points out the offending classes and produces a counterexample, which can be used to reason about the violation. In general, false positives related to

12 We have also tested with a threshold of 40,000 paths and 1.5 GB of maximum heap memory, which eventually resulted in OutOfMemoryError.

Table IV: Summary of detected problems.

Table V: Summary of execution time.

VI. Related Work

A. Equals Design and Implementations

Various authors, including us [17], have investigated design guidelines for equality [13] [5]. Vaziri et al. extend Java with relation types with which equality can be specified in terms of object properties and equals implementation can be generated automatically [21]. The generated equals uses the getClass() API for type checking and does not permit inter-class equality. Rayside et al. also use annotations to specify the abstract functions of an object’s representation and implement equality directly in terms of the abstract functions [15]. We analyze the existing implementation of the equals methods from the available code and do not require any annotations.

B. Program Analysis and Verification

Our checker captures the full call-stack at all control points starting from the entry method. This allows us to symbolically...
gather fairly precise information at each point in a path. In contrast, conventional analyses assume that a path-based approach would not scale and instead, trade off between performance and precision by varying the levels of sensitivities [18].

Saturn is an intra-procedurally path-sensitive and summary based, context-sensitive program analysis framework for C [1] [8]. The summary definition is specific to each checker and property. In contrast, we summarize code behavior using domain specific abstractions inter-procedurally. Das et al. propose a path-sensitive program verification tool called ESP [6] for C. Their approach is similar to ours in that their path-based analysis is guided by domain semantics. They track only branches relevant to the property to be checked (a.k.a inter-procedural property simulation).

Beyer et al. develop a template-based, counter-example guided abstraction inference method [4]. The discovered invariants can then help eliminate infeasible counterexamples. In contrast, we perform code pattern recognition directly from paths to identify key abstractions for model checking.

C. Abstraction Recognition and Modelling

Johnson and Soloway have developed the PROUST tool for interactive analysis and understanding of Pascal programs to support novice programmers [12]. Seemann and Gudenberg propose a technique for detecting design patterns [19]. Their approach of relying on some well-defined names for abstraction is similar to ours. However, our pattern detectors are not limited to names, inheritance or composition. We look at both structural as well as behavioral properties in a path to perform behavioral abstractions. The QBO can answer queries expressed as constraints on an outline model of source code [3]. Different from all these prior work, we use abstraction recognition to create a logic model from code for model checking.

Jackson describes the Alloy environment and method in detail in [11]. However, equivalence relation in particular is not discussed.

D. Existing Checkers

There are several static checkers available for Java. FindBugs 13 handles 36 categories of problems related to equals(). Only four of them deal with equivalence, for example, equals always returning true or false, and the use of instanceof operator in both superclass and subclass implementations. However, these four represent only a small subset of all of the violations that our checker can detect. Hou et al. use SCL to specify and detect violation of implementation constraints for equals() within each individual class [10] [9]. Inheritance and subtyping is not the focus of those work.

VII. CONCLUSION

We have shown how an abstraction-oriented technique can be applied to a specific program analysis problem, that is, equality correctness. We have also shown how we make an inter-procedural, path-based approach scale for this particular problem. In particular, we control path explosion by avoiding method expansion whenever appropriate, and by filtering infeasible paths through path-sensitive data flow analyses. This path-based approach allows for the identification of low-level errors in the paths. More importantly, it provides a basis for contructing a high-level semantic model in Alloy, which is then model checked for semantic errors. Thus, this is essentially a two-layered approach for program analysis. The results of evaluating our checker on three non-trivial projects confirm that this path-based approach can scale up and can be useful. Future work should be directed to generalize the existing checker into an analysis framework and to apply it to other analysis problems.

ACKNOWLEDGMENT

The authors thank Francois Bronsard for discussions of this work and the four anonymous reviewers for the reviews.

REFERENCES


13http://findbugs.sourceforge.net/