Finding Errors from Reverse-Engineered Equality Models using a Constraint Solver

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

***Abstract*—Java objects are required to honor an equality contract in order to participate in standard collection data structures such as List, Set, and Map. In practice, the implementation of equality can be error prone, resulting in subtle bugs. We present a checker called *EQ* that is designed to automatically detect such equality implementation bugs. The key to *EQ* is the automated extraction of a logical model of equality from Java code, which is then checked, using Alloy Analyzer, for contract conformance. We have evaluated *EQ* on four open-source, production code bases in terms of both scalability and usefulness. We discuss in detail the detected problems, their root causes, and the reasons for false alarms.**

***Keywords*-Object Equality; Abstraction Recognition; Path-Based Analysis; Model Finding; Alloy; Soot; Java**

I. Introduction

Software systems rely on various rules to govern the interactions among their components. Such rules are often specified by the API developers and obeyed by the application developers. One important case in Object-Oriented languages such as Java[1] and C#[2] is the contract for the `Object.equals()` method, which requires that all objects satisfy the three properties of the equivalence relation (reflexivity, symmetry, and transitivity) in order to participate in collections such as `List` and `Set`. Breaking this contract often leads to unforeseen bugs that are hard to diagnose even for an experienced developer [4] [21] [23] [26] [10]. Indeed, the correct implementation of equality is probably a concern for all programmers. For example, 622 classes in JDK 1.5 override `Object.equals()`, covering such diverse areas as networking, security, CORBA, RMI, and utilities.

Previously, we conducted an empirical study of the design and implementation of equality using open-source Java projects [22]. We formulated a set of design guidelines for equality, especially in the presence of subtyping and inheritance. With the assistance of a lightweight checker, we also collected an initial set of problematic cases from these projects. Recently, the Lucene community has confirmed that they have fixed the problems we reported for Lucene 2.0 [3]. However, the checker was AST-based, without considering control flow or paths. As a result, it produced too many false alarms to be used as a development tool. However, the presence of true violations in widely used production code motivated us to investigate a new checker, *EQ* [4].

EQ employs a two-layered approach. The first layer performs inter-procedural, path-based data-flow analyses to check for possible low-level errors such as `NullPointerException` and `ClassCastException`. However, its major novelty lies in the second layer, where *EQ* recognizes, directly from Java code, the various abstractions involved in defining equality and expresses them in an Alloy model [15]. Alloy Analyzer is used to detect violations of the equivalence properties from the model.

An example. Consider the `ObjectReferenceTemplateImpl (ORTI)` class shown in Listing 1. This class, a part of CORBA in Java 1.5 (the `com.sun.corba.se.impl.ior` package), implements an abstract value type defined using IDL (Interface Definition Language). *EQ* analyzes the `equals()` methods in the `ORTI` type hierarchy and produces the Alloy model of Listing 2.

Listing 1. *ObjectReferenceTemplateImpl* (renamed to ORTI)

```
1  class ORTI extends ORPB implements ORT, SV{
2   private IORT iort;
3   public ORTI(ORB orb, IORT iort) {
4    super(orb);
5    this.iort = iort;
6   }
7   public boolean equals(Object obj) {
8    if (!(obj instanceof ORTI))
9     return false;
10   ORTI other = (ORTI)obj;
11   return (iort != null) && iort.equals(other.iort);
12  } ...
```

Alloy Analyzer reports a reflexivity violation for Listing 2, with a counterexample shown in Figure 1. The counterexample shows that for an `ORTI` object `a` whose field `iort` is `null` (indicated by the 0 in Figure 1), `a.equals(a)` is false, thus breaking reflexivity. This loss of reflexivity can manifest itself as a bug in the following code, since `ArrayList` assumes that reflexivity hold for `a`:

```
ORTI a=new ORTI(null, null);
ArrayList<ORTI> list=new ArrayList<ORTI>();
list.add(a); // list.contains(a) returns false. A bug!?
```

[1]http://tinyurl.com/java-equals. All URLs verified on April 18, 2011.
[2]http://tinyurl.com/csharp-equals
[3]http://tinyurl.com/lucene-equals
[4]EQ is open-source at http://eqchecker.sourceforge.net.

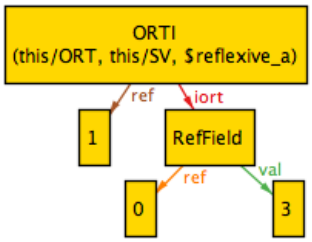

Fig. 1. Alloy Analyzer counterexample for *ORTI* violating reflexivity.

It is also possible that the original programmer intended *iort* never to be null. But then the nullity checking at line 11 of Listing 1 would be redundant and misleading. *EQ* would still be useful in reminding the programmer about this problem.

Listing 2. Full Alloy model for ORTI.

```
1 sig RefField {ref: Int, val:Int}
2 fact {all a,b:RefField|(a.ref=b.ref)=>(a=b)}
3 fun RefField :: equals(that:RefField): Bool {
4  (that.ref!=0 and this.val=that.val) => True
5  else False
6 }
7
8 abstract sig Object {ref : Int}
9 fact{all a,b:Object|(a.ref=b.ref)=>(a = b)}
10 sig SV in Object {}
11 sig ORT in Object {}
12 abstract sig ORPB extends Object {}
13 sig ORTI extends ORPB {iort : RefField}
14 fact {ORTI in ORT}
15 fact {ORTI in SV}
16
17 pred Object :: equals(that: Object) {
18  (this in ORTI) =>(
19  (that in ORTI) and (that.ref!=0) and
20  (this.iort.ref != 0) and (this.iort.equals[that.iort] != False))
21 }
22 // spec. of common equivalence properties
23 assert reflexive{all a:ORTI | a.ref!=0 => a.equals[a]}
24 assert symmetric{all a,b:ORTI | (a.ref!=0 and
25  b.ref!=0) => (a.equals[b] <=> b.equals[a])}
26 assert transitive{all a,b,c:ORTI | (a.ref!=0
27  and b.ref!=0 and c.ref!=0) => (a.equals[b]
28  and b.equals[c] => a.equals[c])}
29 assert nullity{all a: ORTI | (a.ref != 0) =>
30  (all b:ORTI | (b.ref=0) => not a.equals[b])}
31 // finding counterexample within small scopes
32 check reflexive for 1
33 check symmetric for 2
34 check transitive for 3
35 check nullity for 2
```

Overview of approach. Because objects from the same type hierarchy may be compared for equality, *EQ* creates an Alloy model for each type hierarchy that contains user-defined `equals()`. The Alloy model expresses the equality logic for the entire hierarchy with a single predicate, which is also named `equals()` (see Listings 2 and 5 for examples).

In general, the body of the `equals()` predicate consists of a set of mutually exclusive implications, each of which defines the equality logic for a concrete class that uses an overridden `equals()`. *EQ* recognizes and expresses equality-related abstractions as Alloy predicates, from the inter-procedural paths of an `equals()` method. For example, the type testing at line 8 and the statement at line 11 in Listing 1 are translated to the predicates at lines 19 and 20 of Listing 2, respectively. All equality-related abstractions on the same true-returning path are joined with logic conjunction, which are further joined as disjunction.

Our evaluation using four open-source projects shows that *EQ* produces a decent rate of false alarms (under 28%, Table III). Moreover, although theoretically unsound, for the four projects evaluated, *EQ* practically produces zero false negative. Unlike closely related work such as [10], *EQ* not only detects errors but also produces human-readable formal documentation, for example, Listings 2 and 5.

The remaining paper is organized as follows. Section II describes how *EQ* models Java in Alloy. Section III presents the algorithms for detecting equality-related abstractions from code. Section IV discusses the generation of inter-procedural paths. Results of running the checker on benchmarks are presented in Section V. Section VI compares *EQ* with related work. Finally, Section VII concludes the paper.

II. Modeling Java in Alloy

EQ creates an Alloy model from Java code. This section presents how *EQ* models Java constructs using Alloy.

A. Types and Inheritance

We model Java types with Alloy signatures. An abstract class is modeled with an abstract signature. Class inheritance is modeled as signature extension. Implementing interfaces creates multiple inheritance, which is modeled using assertions and subsetting with the `in` keyword (see [15], pp. 94).

As an example, consider the Alloy model in Listing 2 for the `ORTI` class in Listing 1. `Object` is the root of the type hierarchy (line 8). `SV` and `ORT` are two interfaces declared as subtypes of `Object` using the `in` keyword (lines 10 and 11). `ORTI` extends the abstract signature `ORPB` (lines 12 and 13). Finally, the fact that `ORTI` implements both `ORT` and `SV` are modeled by the two Alloy facts at lines 14 and 15.

B. Fields and Nullity

A field in a Java type is modeled as a field in the corresponding Alloy signature. Fields of primitive Java types are modeled as Alloy `Int`. Fields of reference types are modeled as Alloy signature `RefField` (see lines 1 and 13 of Listing 2) with two aspects: the reference and the value. The reference is modeled as Alloy `Int`, with 0 representing `null`. The value of a standard collection type such as `Set` and `Map` is modeled with their counterparts in Alloy. When checking `equals()` in the current type hierarchy, we assume that other type hierarchies have implemented equality correctly. So we model values of non-collection types as `Int`.

C. Unexpanded Function Calls

EQ does not expand some "standard" functions called by `equals()`. A call of a standard `equals()` on a field is converted to an equality predicate in Alloy. An

example of this behavior can be found in Listing 2 (lines 3-6 and 20). Calls to the standard `compareTo()` method is processed similarly. Finally, standard arithmetic and logical operators are modeled as Alloy functions available in the `util/integer` and `util/boolean` modules. For instance, `this.f+5==that.f+5` is translated to `Add[this.f,5]=Add[that.f,5]`.

When the semantics of an invoked method is unknown, we model it as a nondeterministic function. In particular, this applies to the bitwise operators, which are not supported by Alloy. Listing 3 provides an example from JDK 1.5, where a `Principal` represents a host, and a `Group` represents a group of hosts in the same subnet. The `PrincipalImpl (PI)` and `GroupImpl (GI)` classes implement them. The `hashCode() (hc)` method returns an integer representation of the principal's IP address. A bitwise `and` operator is used to check if `this` subnet is *part of* `that` subnet. Clearly, this example violates the symmetry property, as revealed by the following code:

```
GI group1 = new GI(''192.168.1.128'');
GI group2 = new GI(''192.168.1.0'');
group1.equals(group2); // Returns true
group2.equals(group1); // Returns false
```

The 32-bit IP addresses for the two groups differ only in the last byte. Since $128 \ \& \ 0 = 0$ but $0 \ \& \ 128 \neq 128$, symmetry is broken. *EQ* detects this violation from Listing 4.

Listing 3. *equals()* of *com.sun.jmx.snmp.IPAcl.GroupImpl*.

```
1  class PI implements Principal ... {
2   private InetAddress[] add = null;
3   public PI(String hostName) {
4    ...
5    add = InetAddress.getAllByName(hostName);
6   }
7   public int hashCode() {
8    return add[0].hashCode();
9   }...
10 class GI extends PI implements Group ... {
11  public boolean equals (Object p) {
12   if (p instanceof PI || p instanceof GI){
13    if ((super.hashCode() & p.hashCode()) == p.hashCode())
14     return true;
15    else return false;
16   } else {return false;}
17  }...
```

For this example, *EQ* also reports violations of reflexivity and transitivity, which are false alarms. These are due to the fact that `BitwiseAnd` is modeled as a nondeterministic function (lines 1-4, Listing 4). For an integer a, the bit-and operator guarantees that $a \ \& \ a = a$. But all that `BitwiseAnd` specifies is that it takes two *Int* as parameters and returns a single *Int* value, which is a weaker semantics. If Alloy had supported the bit-and operator, these two false alarms would not have existed.

Listing 4. Alloy model for *GroupImpl* (Listing 3).

```
1 lone sig Relation {r: Int -> Int -> one Int}
2 fun BitwiseAnd(t1: Int, t2: Int): Int {
3    t2.(t1.(Relation.r))
4 }
5 abstract sig Object {ref : Int, hc : Int}
6 // Rest of type hierarchy elided
7 pred Object :: equals( that: Object ) {
8  (this in GroupImpl) =>
9  ((that in PrincipalImpl) and
10 (BitwiseAnd[this.hc, that.hc] = that.hc))
11 }
```

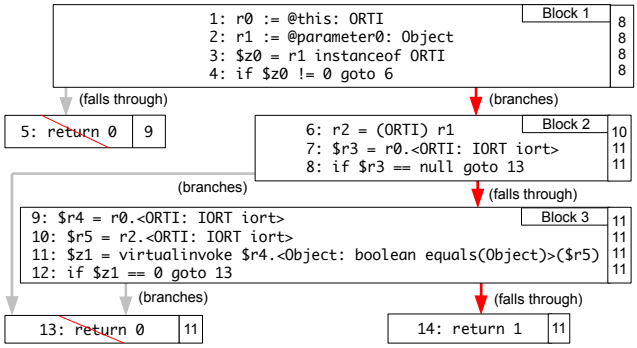

Fig. 2. The flow graph of *ORTI.equals()* in Jimple (The numbers on the right-hand side are for Java statements).

III. Recognizing Equality-related Abstractions

The main input to *EQ*'s abstraction detectors is the interprocedural paths produced from Soot's Jimple control flow graphs [25]. A path-sensitive copy propagation analysis is used to track the data flow on a path, and an expression tree is built for each Jimple statement in order to recognize the high-level abstractions that the statement may contain.

EQ currently recognizes six abstractions, i.e., *type testing* and five equality comparisons (of both simple *state* as well as data structures such as *array*, *list*, *set*, and *map*). These six detectors can adequately handle the `equals()` methods in our evaluation projects, but new detectors can be added for additional patterns or abstractions. In this section, we discuss three in detail.

A. Type Testing and State Comparison

The type testing detector converts four kinds of type expressions into Alloy predicates: `o instance- of T`, `(T)o`, `o.getClass()==T.class`, and `this.getClass()==o.getClass()`. To illustrate, consider the true-returning path in Figure 2. The detector iterates through the statements on the path to check whether an `If` statement has one of the above four type expressions as its condition. In this case, it finds the one in block 1. The translation is achieved by constructing an expression tree (Figure 3) for the conditional expression (`$z0!=0`), which is converted into an Alloy predicate by considering the comparison operator (`!=` or `==`), its right hand side operand (`0` and `1` representing `true` and `false`, respectively), and the control flow edge (*branches* or *falls through*) taken on the path. Since the `instanceof` also tests the non-nullity of `that`, the fact about non-nullity is also added in Listing 2, along with the fact about type testing (line 19).

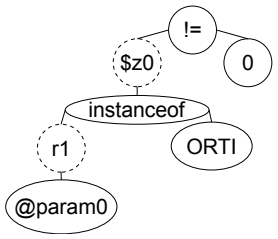

Fig. 3. Expression tree for `$z0!=0`.

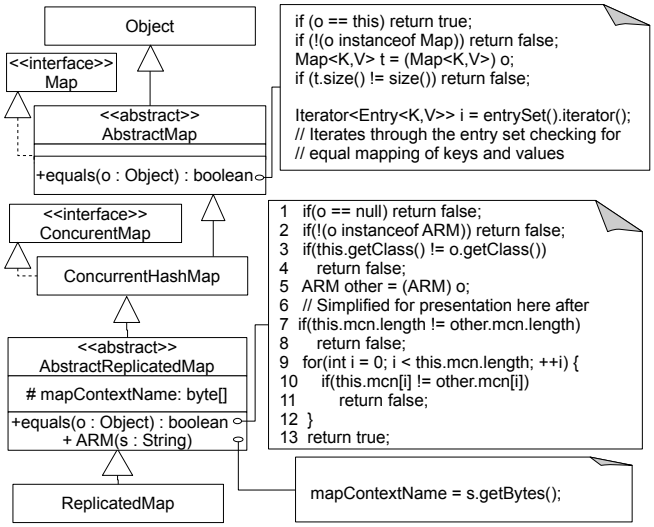

Fig. 4. *AbstractReplicatedMap* and its hierarchy (simplified).

The state comparison detector is responsible for the inequality comparisons at line 20 in Listing 2, which are generated for the `if` statements in blocks 2 and 3 of Figure 2, respectively. The type testing and state comparison detectors are enough to translate the `equals()` in Listing 1 into Alloy.

B. Array Comparison

To illustrate the detection of array comparison, consider the `equals()` method for `AbstractReplicatedMap (ARM)` in Figure 4. `ARM` from Tomcat 6.0 implements a map that is sharable in a cluster through an RPC (Remote Procedure Call) channel. Such maps are distinguished by the value of `mapContextName (mcn)`. Thus, `ARM.equals()` overrides `AbstractMap (AM)` and re-defines its equality in terms of `mcn`. In particular, it compares two arrays in lines 7-12. Two paths are produced: `[1,2,3,5,7,9,13]` and `[1,2,3,5,7,9,10,9,13]`.

To detect an array comparison, the detector searches for the presence of the following four features on a path, where the first three are critical and the last one optional:

- **Element Comparison** requires the presence of a statement that compares array elements (line 10), and the arrays be fields of `this` and `that`. Furthermore, it is also required that the index expressions must share some variables. Only the second path passes this test.
- **Loop Header** requires the presence of a loop header [2] on the path. The loop header must be an inequality expression. In addition, it must refer to a local variable that is shared by the indices of the element comparison (e.g., the index variable `i` in lines 9 and 10).
- **Loop Body** requires that the element comparison be contained by the same loop as the loop header.
- **Length Comparison** represents a statement that compares the lengths of two arrays (line 7).

We distinguish two kinds of features: *critical* and *optional*. All critical features must be found on a path to infer the presence of an array comparison. Optional features, on the other hand, are not absolutely needed but their presence would enforce the inference. The checker issues a warning for each missing optional feature.

The final Alloy model for the entire type hierarchy of `ReplicateMap` is shown in Listing 5, where `RM` represents `ReplicateMap`, `ARM AbstractReplicateMap`, and `CHM ConcurrentHashMap`. The `for` loop for array comparison in `AbstractReplicatedMap` is abstracted into the comparison of Alloy sequences at line 17. `ArrayField` is used by `ARM` to represent `mcn` and `MapField` (`Map_map`) by the `Map` interface to represent its data abstraction.

Listing 5. Alloy model for the hierarchy in Figure 4.

```
1  sig ArrayField {ref: Int, val: seq Int}
2  sig MapField {ref: Int, val: Int −> lone Int}
3  // Nullness and reference test facts elided
4  sig Map in Object {Map_map : MapField}
5  abstract sig ARM extends CHM {
6   mcn : ArrayField
7  }
8  ...
9  pred Object :: equals( that: Object ) {
10 (this in CHM − ARM) => (
11   (that.ref = this.ref) or
12  ((that in Map) and (that.ref != this.ref) and
13  (this.Map_map.val = that.Map_map.val)))
14 else
15 (this in RM) => (
16  (that in RM) and (that.ref != 0) and
17  (this.mcn.val = that.mcn.val))
18 }
```

This example also uses the map comparison detector to recognize the equality logic in `AbstractMap.equals`, which implements two ways of comparing maps and inherited by `ConcurrentHashMap`. The first path just compares `this` with `that`, which produces the equality comparison at line 11 of Listing 5. The other paths check if the two maps contain equal key-value pairs, which is abstracted as a value comparison in Alloy (lines 12-13).

Alloy Analyzer reports violations of both symmetry and transitivity for Listing 5. The symmetric violation is caused by the `getClass()` statement used by `Abstract-ReplicatedMap`, which allows only objects of the same type to be equal (line 3, Figure 4). `AbstractMap`, on the other hand, uses `instanceof`, allowing any two `Maps` to be equal if they have equal key-value pairs.

The transitivity violation is caused by the addition of a new aspect in `ARM`. `ConcurrentHashMap` defines equivalence by using the default key-value pairs, but `ReplicatedMap`

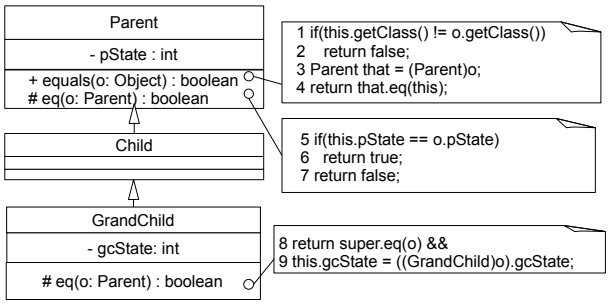

Fig. 5. *equals()* in a type hierarchy.

uses `mapContextName`. In case that the intention is to use `ARM` independent of `ConcurrentHashMap`, our checker can still serve as a reminder to the programmer that they should confirm and perhaps explicitly document this assumption.

IV. Path Generation

EQ starts by searching a Java project for `equals()` methods that override `Object.equals()`. The type hierarchy that contains each `equals()`, excluding `Object`, is constructed. *EQ* analyzes the `equals()` once for each of its receiver classes that may impact its semantics, for example, when a subclass overrides a method the `equals()` method calls. A path enumeration algorithm is applied to the control flow graph of the `equals()` method to produce intra-procedural paths. Method calls on an intermediate path are expanded to produce inter-procedural paths.

A. Loop Unrolling

The key to our loop unrolling strategy is to produce paths where a loop condition is evaluated to the same truth value at most once. This strategy generates a minimal set of paths while still preserving the semantics of the original control flow graph. For example, consider the following snippet:

B1
while (C1||C2) B2;
B3

where `C1||C2` is the loop condition, `B1`, `B2`, and `B3` represent basic blocks, and `C1` and `C2` simple boolean expressions such as `i < 0`. The following set of paths would be produced:

1 B1, [C1,F], [C2,F], B3 (valid)
2 B1, [C1,T], B2,[C1,F], [C2,F], B3 (valid)
3 B1, [C1,F], [C2,T], B2, [C1,F], [C2,F],B3 (valid)
4 B1, [C1,T], B2, [C1,T] (pruned)
5 B1, [C1,T], B2, [C1,F], [C2,T] (pruned)

where `T` represents the `true` branch and `F` the `false` branch. The first three paths are valid according to our strategy. The fourth path is pruned because it evaluates the loop condition `C1||C2` to `true` twice (`[C1,T]` followed by `[C1,T]`). The fifth path is pruned for the same reason (`[C1,T]` followed by `[C1,F],[C2,T]`).

B. Virtual Method Resolution and Expansion

Once a (minimal) set of intra-procedural paths is extracted, the path generation algorithm iterates through each path looking for a method call for further expansion. The algorithm expands recursive calls only once. Consider the `equals()` in Figure 5. Two intra-procedural paths are produced for `Parent.equals()`: `[1,2]` and `[1,3,4]`. Since equality is a predicate, only `true`-returning paths are kept, and `false`-returning paths such as `[1,2]` are pruned.

For path expansion, consider path `[1,3,4]` as an example. In general, we do not expand method calls that have predefined, standard semantics. Instead, *EQ* directly uses their standard semantics in further analyses. The two `getClass()` calls in line 1 are such examples. Other method calls that are not expanded include:

- standard methods such as those from the Collection Framework and `hashCode()` (`equals()` and `compareTo()` get expanded when they are invoked on `this` or `that`);
- method calls on a field (*EQ* also provides an option for a user to specify the meaning of a custom method);
- the equality comparison between two calls of the same method, i.e., `this.getF()==that.getF()`, and `StaticM(this.f)==StaticM(that.f)`.

The virtual call `that.eq(this)` at line 4 of path `[1,3,4]` is expanded as follows. Because our flow analysis tracks the types of a variable along a path, it infers that the type of `that` at line 4 can only be `Parent`. Thus, `Parent.eq()` is the only resolution target for the call. Using the regular CHA, however, the call would have been resolved to both `Parent.eq()` and `GrandChild.eq()`. The body of `Parent.eq()` is further inlined into `[1,3,4]`, resulting in two paths: `[1,3,4,5,6,4]` and `[1,3,4,5,7,4]`. In both paths, line 4 is repeated twice to represent the method entry and the method exit, respectively. The second path is pruned as it returns `false`.

C. Path Filtering

To alleviate the path explosion problem and to improve performance, four data-flow-based filters are used to prune both false-returning paths and infeasible paths:

- **Boolean Filter** prunes both false-returning paths and paths that have conflicting values for a boolean variable.
- **Nullity Filter** prunes paths that have a conflict in the `null` value for a reference variable.
- **Type Filter** prunes paths that have a conflict in the types of a variable.
- **Throw Filter** prunes paths that end with an exception throwing node as we do not model exceptions in Alloy.

We handle the exception handling control flow within a method but not exceptions that escape a method boundary. The latter is not critical for our problem.

V. Evaluation

EQ was evaluated using four popular open source projects for usefulness, accuracy, and performance: Tomcat 6.0 (T6),

TABLE I
CHARACTERISTICS OF THE EVALUATION PROJECTS.

	T6	L3	JF1	J5
Interfaces	175	76	128	1745
Classes	1236	888	866	11240
Type hierarchies	23	31	163	427
Defined equals	29	96	377	622
Processed equals	36	108	409	1115
Expanded methods	102	199	839	26597

TABLE II
SUMMARY OF DETECTED ABSTRACTIONS.

	T6	L3	JF1	J5
Identity equality	35	247	756	1861
Type testing	58	225	733	3814
State comparison	164	398	2036	9933
Array comparison	1	6	7	56
List comparison	0	0	0	36
Set comparison	0	0	0	4
Map Comparison	1	0	0	31
Unexpanded methods	82	240	7008	3475
Total	**341**	**1116**	**10540**	**19210**

Lucene 3.0 (L3), JFreeChart 1.0.13 (JF1), and JDK 1.5 (J5). Table I summarizes their characteristics. The third row shows the number of type hierarchies that override `equals()`. The number of processed `equals()` is greater than that of the defined ones, because when a subtype inherits an `equals()` from a supertype and overrides a method called by that `equals()`, the `equals()` must be processed again in the context of the subtype. The last row depicts the number of methods expanded.

Table II summarizes the various abstractions *EQ* detected in the four projects. The first row shows the number of reference equality expressions of the form `this==that`, and the last row the number of methods that were modeled directly in Alloy without expansion. JDK contains significantly more comparisons of data structures than others.

A. Detected Problems

Table III depicts the categories of violations that *EQ* detected in the four projects. We have manually verified all of the reported violations. We discuss these categories, their root causes and solutions, and the reasons for false alarms.

1) Non-symmetric null testing: The equality predicate should be defined symmetrically, involving the same states from both `this` and `that`. Otherwise, it may result in contract violations. One example is the violation of reflexivity for the example of Listing 1, which can be fixed by considering the two objects equal when both `other.iort` and `this.iort` are `null`.

2) Improper type testing in hierarchy: This category concerns problems caused by the inappropriate type testing in a type hierarchy. We have identified three kinds of type hierarchies that may impact equality in [22]: *type-compatible*, *type-incompatible*, and *hybrid*. A type-compatible hierarchy allows objects of a parent type to be equal to those of a subtype (implemented using `that instanceof Parent`).

TABLE III
TRUE ERRORS (E), CAUSES, AND FALSE ALARMS (FA).

violations by causes	T6		L3		JF1		J5	
	E	FA	E	FA	E	FA	E	FA
Null testing	1	0	0	0	0	0	5	0
Type testing	1	0	2	0	12	0	12	0
Multiple criteria	1	0	0	0	4	0	12	0
Op. comparison	0	0	0	0	0	0	2	2
Typos	0	0	0	0	2	0	1	0
Empty paths	0	0	0	0	0	0	3	0
Adapters	2	4	0	0	0	0	19	18
Ovrldng smlrty	4	0	0	0	0	0	16	0
Ovrldng ovrdng	4	0	0	0	0	0	28	0
Null parameters	0	0	7	0	0	0	13	0
ClassCstException	0	0	2	0	1	0	17	2
Other FA's	0	0	0	0	0	0	0	33
Total	**13**	**4**	**11**	**0**	**19**	**0**	**128**	**55**

A type-incompatible hierarchy will not allow objects of a parent type to be equal to a subtype (implemented using `this.getClass()==that.getClass()`). A hybrid hierarchy allows both type-compatible and incompatible sub-hierarchies within the same type hierarchy while still respecting the equality contract.

When the type testing statements are used carelessly, we may end up with a hierarchy that is none of the above, resulting in violations of symmetry or transitivity. An example is the type hierarchy of `AbstractReplicatedMap` in Figure 4. The `Map` interface implicitly specifies a *type-compatible* hierarchy with its behavioral specification. `AbstractMap` respects this specification by performing the `that instanceof Map` test in its `equals()`. `AbstractReplicatedMap`, however, breaks this design by using `getClass()` (line 3), thus starting a new type-incompatible sub-hierarchy. The `Map` hierarchy is not implemented to accommodate this behavior, resulting in a violation of symmetry. Section III-B explains the cause for the violation.

The problem can be fixed in two ways: by implementing the `Map` hierarchy as a hybrid hierarchy (details in [22]) or just making `AbstractReplicatedMap` compose, rather than extend, `ConcurrentHashMap`.

3) More than one criterion for equality: When two objects are compared using different criteria, the equality logic may violate symmetry or transitivity. Unfortunately, we have seen these violations regardless whether type hierarchies are involved. They happen when an `equals()` defines multiple true-returning paths that check different states for equality, which should always be avoided. An example that involves a type hierarchy is the transitivity violation between `ConcurrentHashMap` and `ReplicatedMap` in Section III-B (Figure 4). The solution is again to either use a hybrid type hierarchy [22], or use composition instead of implementation inheritance [4].

4) Non-symmetric comparison of operations: Sometimes, the return value from an operation on a field is used as an aspect for checking equality. When the same operation is applied symmetrically on fields, such as

`f(this.x)==f(that.x)`, it is safe. Otherwise, it may result in violations of symmetry or transitivity. For example, line 13 of Listing 3 performs non-symmetric comparison of operation on fields and, thus, violates symmetry, as elaborated in Section II-C.

5) Typographical errors: Programmers make typographical errors. A compiler can help spot syntactic errors easily but not the semantic ones. Consider the `WindDataItem` class from JFreeChart in Listing 6. Instead of returning `true` for the identity comparison at line 2, the code returns `false`, which results in a reflexivity violation.

Listing 6. *org.jfree.data.xy.WindDataItem.equals().*

```
1 public boolean equals(Object obj) {
2  if (this == obj) {return false;} ...
```

6) Empty paths: By default, `Object.equals()` implements reference equality, under which an object can be equal to only itself. The body of `equals` of the `java.net.InetAddress` class, however, contains only '`return false;`'. Thus, objects of this class cannot be equal to anything, not even itself. This class is the super class for the IPv4 and IPv6 Internet addresses in JDK 1.5. As all of the paths for the method are pruned, the checker detects this situation and reports a violation of reflexivity.

7) Improper adaptation: A key feature of the adapter design pattern is that the *adapter* subtypes its *adaptee*. In the absence of this subtyping, object adaptation is prone to errors when equality is concerned.

Consider the `equals()` of the `Token` class shown in Lisiting 7 from JDK 1.5. When an object of `Token` is passed to the `equals()`, it invokes the `equals` method of `String` at line 6. Since `Token` is not a subclass of `String`, the call at line 6 always returns false. This results in a violation of reflexivity. Two solutions are possible. The first is to implement the adapter pattern correctly. However, if adaptation is not intended, the similarity implementation discussed in the next subsection can be considered.

Listing 7. *sun.tools.jconsole.inspector.XTree.Token.*

```
1 class Token{
2 private String token;
3 public Token(..., String token) {...}
4 public boolean equals(Object object){
5  if (object instanceof Token)
6  { return token.equals(((Token) object));}
7  else {return false;}
8 }}
```

Of the 37 violations reported in this category, 18 are false alarms (1 symmetry from this example and 17 transitivity). All false alarms are caused by the imprecise modeling of the `equals` method calls. More specifically, when the receiver and argument for a call to `equals()` do not belong to the same type hierarchy, the `equals()` is modeled as a non-deterministic function (see Section II-C), for example, the `equals()` call at line 6 in Listing 7.

8) Overloading for similarity: Overloading `equals` may cause unintended problems. So, in addition to checking `equals(Object)`, *EQ* also detects the special cases of overloaded `equals()` (this subsection) and overloading but without overriding (next subsection).

Listing 8. Overloaded *sun.tools.tree.StringExpression.equals().*

```
1 String value;
2 public SE(..., String value) {...}
3 public boolean equals(Object obj) {
4  if ((obj != null) && (obj instanceof SE)) {
5   return value.equals(((SE)obj).value); }
6  return false;
7 }
8 public boolean equals(String s)
9 {return value.equals(s);}
```

Programmers often overload `equals()` to compare two objects that do not belong to the same type hierarchy. Consider the overloaded `equals` in `StringExpress- ion (SE)` from JDK 1.5 (Listing 8). Overloading can make a `StringExpression` equal to a `String` but not the reverse, thus breaking symmetry. Liskov and Guttag [19] suggest that the equality test should be reserved only for types belonging to the same type hierarchy. If two objects need to be compared for similarity, a method with a different name should be used, such as `String` and `StringBuffer` in JDK.

9) Overloading but without overriding: The `equals` method has a special purpose of comparing two objects for equality. Overloading but without overriding this method can cause unintended side effects (also see Item 26 in [4] and Puzzle 58 in [5]). Consider the overloaded `equals()` in Listing 9 from JDK. It returns true when compared with any `Permission` object. But this class does not override `equals(Object)`. So, when an object of a type other than `Permisson` is compared, the default reference equality from `Object`, instead of the overloaded `equals()` at line 3, is called. This behavior may not be what is intended.

Listing 9. Overloaded *sun.security.acl.AllPermissionsImpl.equals().*

```
1 class AllPermissionsImpl extends ... {
2 public AllPermissionsImpl(String s) {...}
3 public boolean equals(Permission another)
4     {return true;}
5 }
```

10) Null parameter: Besides the three properties for equivalence, the equality contract also requires that the method return `false` when `null` is passed in. The `FontLineMetrics` class of JDK 1.5 in Listing 10 does not respect this contract when `rhs` is `null` as it would throw `NullPointerException`.

Listing 10. *sun.font.FontLineMetrics.equals().*

```
1 public final boolean equals(Object rhs){
2  try{
3   return cm.equals(((FontLineMetrics)rhs).cm);
4  }catch (ClassCastException e) {return false;}
5 }
```

Listing 11. *PayloadAttributeImpl.equals().*

```
1 public boolean equals(Object o) { ...
2  if (o instanceof PayloadAttribute) {
3   PayloadAttributeImpl that = (PayloadAttributeImpl)o; ...
4  } ...
5 }
```

11) Probable ClassCastException: The `equals()` method almost always casts its parameter from `Object` to some type. There can be a mismatch between the type casting and other type testing expressions. Consider `PayloadAttributeImpl.equals()` in Listing 11 (`org.apache.lucene.analysis.tokenattributes` package, Lucene 3.0). `PayloadAttribute` is a super interface for both `PayloadAttributeImpl` and `org.apache.lucene.analysis.Token`. Thus, if an object of `Token` is passed in, a `ClassCastException` will be thrown at line 3. The problem can be fixed by replacing `PayloadAttributeImpl` with `PayloadAttribute`. Furthermore, the two false alarms listed for this category are caused by an internal bug related to the way JDT computes type hierarchies.

12) Other false alarms: As discussed in Section II-C, when *EQ* does not know the semantics of a method, and expanding that method would not necessarily lead to a better Alloy model, we choose to model it as a nondeterministic function in Alloy. Although weaker than the full semantics, a nondeterministic function would be the best that the checker could do. Not surprisingly, modeling a non-standard, unexpanded method as a nondeterministic function turns out to be a major cause for false alarms. The solution is to provide a way for a user to specify the meaning of such a function to the checker or to apply some heuristics to infer the semantics of such a method to reduce false alarms.

B. Evaluation of Un-handled Cases

EQ handles the majority, but not all, of the `equals()` in the four evaluation projects. Knowing more about these obstacles may help improve the checker. To that end, we have carefully analyzed the 57 cases in JDK 1.5 that *EQ* was unable to handle. We conclude that 42 of the 57 cases could be handled by a straightforward extension of the checker, and 15 pose unique challenges that needs further research. Five cases could not be handled due to path explosion (Section V-D). Cases that can be supported are:

- *Implementation variations of standard abstractions (10 cases).* For instance, `com.sun.jmx.snmp.SnmpOid` implements an array comparison by checking that the array index following the common prefix of the two arrays has reached the end. These can be handled by adjusting the current array comparison detector.
- *Comparison of two Java enumerations (7 cases).* Comparison of enumerations can be handled similarly as other existing data structure comparisons such as List.
- *Comparison of two multi-dimensional arrays (3 cases).*
- *Comparison of two linked lists (2 cases).*
- *Implicit abstractions (7 cases).* Arrays may be used to implement abstractions such as a set rather than a sequence. For instance, `java.util.BitSet` compares two arrays for set equality.
- *On-the-fly abstractions (5 cases).* Some `equals()` creates and then compares new objects for equality.
- *Collection-like-operations (7 cases).* Classes such as `javax.swing.text.TabSet` exposes a set of methods that essentially make them a sequence. However, these methods are custom made rather than inherited from a standard type such as `List`. Their `equals()` actually implements a comparison of two sequences like an array. Such cases can be handled using function modeling.
- *Debugging logic (1 case).* Code for debugging needs to be separated from equality implementation. A control dependency analysis could be used to achieve this.

For the 15 cases that pose challenges, 7 implement `equals()` in ways that are too complicated for the checker to recognize any meaningful abstraction. The other 8 cases, e.g., the `com.sun.jndi.ldap.SimpleClientId` class, contain logic conditional on the type of a field. We do not know how to properly model such fields in Alloy.

C. Evaluation of False Negatives

Since *EQ* infers higher-level abstractions based on structural patterns of a program rather than its behavior, in theory, it is unsound and may produce false negatives. To evaluate the practical implications of this issue, we have inspected all of the type hierarchies from Tomcat and Lucene, and a random subset of thirty type hierarchies for each of JFreeChart and JDK. As a result, we have found no false negative for cases that *EQ* reports being correct. That is, when the checker says that the equality in a type hierarchy is correct, the equals() implementations in that type hierarchy is indeed correct. Thus, this shows that the theoretical unsoundness of *EQ* might not be an actual problem in practice. Note that we have not counted the unhandled cases reported in Section V-B as false negatives.

D. Performance Evaluation and Path Explosion

The four benchmarks were evaluated on an *iMac* machine (1.6 GHz, 64-bit, quad core). Eclipse was allocated 2 GB of initial and 5 GB of maximum memory through VM arguments. In addition, we have tested that JDK 1.5 can be analyzed with 1 GB of maximum memory.

Table IV depicts a summary of path reduction and overall performance. The total paths include both the final, true-returning paths and the intermediate paths that were filtered. The first five rows depict the numbers of paths each path pruning strategy filtered. Overall, these on-the-fly path pruning strategies have reduced significantly the number of paths, as shown by the ratios between final paths and total paths.

Table IV also depicts the time the checker completed the analysis of each project. On average, about 60% of the time was spent on analyses and model generation, and 40% on model finding by Alloy. It appears that the performance is reasonably efficient for projects of this scale.

To investigate path explosion, we set a threshold of 8,000 paths per `equals()` for the benchmarks. Only five

TABLE IV
SUMMARY OF PATH REDUCTION AND PERFORMANCE.

	T6	L3	JF1	J5
Loop Unrolling	18	30	127	13027
Boolean Filter	285	711	8058	37440
Nullness Filter	0	0	0	1311
Type Filter	0	0	0	122
Throw Filter	0	0	4	1022
Recursion	0	0	0	2
Final Paths	**82**	**323**	**1200**	**9010**
Total	**385**	**1064**	**9389**	**61934**
Path Reduction	**78.70%**	**69.64%**	**87.22%**	**85.45%**
Total Time	**25s**	**51s**	**12m44s**	**29m54s**

`equals()` from JDK 1.5 crossed this limit. The `equals()` from the three other projects were well below the 8,000 limit. For JDK 1.5, the mean and median of the path distribution are 75 and 5, respectively. The 99^{th} and 95^{th} percentile are 1,469 and 71, respectively. This shows that a threshold of 1,500 paths would handle more than 99% of `equals()` in the four projects. We conclude that the path-sensitive approach that we have adopted for the equality checking problem can scale up for realistic projects.

VI. Related Work

Equals Design and Implementation Various authors have investigated design guidelines for equality [19] [4] [22]. 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 [26]. The generated *equals* uses only *getClass()* for type testing and, thus, does not permit inter-class equality. Rayside et al. [21] and Grech et al. [12] use annotations to specify the abstraction functions of an object's representation and implement equality directly in terms of the abstraction functions. We analyze existing *equals* methods, rather than annotate or extend Java.

Marinov and Khurshid propose VAlloy, as an extension to Alloy, to model virtual calls and dynamic dispatching more naturally [20]. They show how `equals()` can be modeled and checked with Alloy Analyzer. They also provide specifications for parts of the Java Collections Framework. But multiple inheritance and reference nullity were not covered, and unlike *EQ*, VAlloy was not implemented [20].

An earlier version of *EQ* is presented in [23], but with a much lighter evaluation. A thorough evaluation of effectiveness is necessary since our approach utilizes heuristics in recognizing abstractions. Furthermore, the current paper also presents several important additions (function modeling, Section II-C and nullity modeling, Section II-B) and improvements (modeling multiple inheritance, Section II-A) since [23]. Loop unrolling has been improved (Section IV-A) so as to prune more paths (Table IV), as a result, six more `equals()` in JDK not handled previously due to path explosion, are now handled. A new evaluation of path explosion appears in Section V-D.

Abstraction Recognition The core of our approach is recognizing abstractions by analyzing path structures. Abstraction recognition, in general, can have many potential applications, for example, to diagnose a novice's program and provide feedback interactively [16], and to generate summary comments for methods [24]. A (partial) survey of abstraction recognition techniques can be found in [1].

Program Verification Bounded program verification techniques (PV) encode either programs [10], [27] or path conditions as logic formulae [8], [9], [17], and solve the verification problem with a constraint solver or a theorem prover. Other PV techniques include theorem-prover-based approaches such as ESC/Java [11] and model checkers [3], [6], [7], [13]. Model checkers check temporal program properties whereas we work on structural properties of equality.

EQ recognizes higher level abstractions than the logical encoding that a PV technique would produce, especially when composite data structures such as arrays and containers are compared. PV's logical encoding can be hard to read [9], [10], [27], while *EQ* essentially reverse-engineers an Alloy logic model that also serves as a form of human-readable documentation. PV can be more general (checking any strong properties for a function), but may become intractable, while *EQ* focuses on a specific behavioral contract and is scalable. PV requires specifications for each checked function, while *EQ* is a push-button technique without this requirement. *EQ* recognizes abstractions directly from a path by analyzing its structure. Thus it is sufficient to unroll each loop only once. *EQ*'s focus on recognizing abstractions allows it to scale up. Khurshid and Suen have made similar observations, recommending directly modeling standard data structures abstractly when verifying client code [18].

Existing Checkers Existing static checkers such as JLint [5], PMD [6], and ECS/Java [11] do not check the equivalence relation. FindBugs [14] handles 36 categories of problems related to `equals()`, but only four of them are related to the equivalence relation, for example, `equals` always returning true or false, and the use of `instanceof` operator in both `superclass` and `subclass` implementations. However, these four comprise only a small subset of all of the violations that our checker can detect.

VII. Conclusion

We describe an automated approach for reverse engineering and checking the correctness of Java equality. Our approach recognizes equality-related abstractions from Java code to form an Alloy model. We have implemented and evaluated a checker named *EQ* on four open-source Java projects in terms of soundness, accuracy, usefulness, and scalability. We present a classification of the root causes for the detected violations and their solutions as well as the causes for false alarms. We also report a thorough assessment of cases that our checker currently does not handle. We conclude that this abstraction recognition approach for checking the equality contract is accurate enough, can scale up to realistic projects and, thus, useful.

[5]http://artho.com/jlint/

[6]http://pmd.sourceforge.net/

REFERENCES

[1] S. K. Abd-El-Hafiz and V. R. Basili, "A knowledge-based approach to the analysis of loops," *IEEE Trans. Software Eng.*, vol. 22, no. 5, pp. 339–360, 1996.

[2] 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.

[3] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," *SIGPLAN Not.*, vol. 37, pp. 1–3, 2002.

[4] J. Bloch, *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

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

[6] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: a new Symbolic Model Verifier," in *CAV*, 1999, pp. 495–499.

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from java source code," in *ICSE*, 2000, pp. 439–448.

[8] X. Deng, J. Lee, and Robby, "Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *ASE*, 2006, pp. 157–166.

[9] G. D. Dennis, "A relational framework for bounded program verification," Ph.D. dissertation, MIT, MA, USA, 2009.

[10] J. Dolby, M. Vaziri, and F. Tip, "Finding bugs efficiently with a SAT solver," in *ESEC-FSE*, 2007, pp. 195–204.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *PLDI*, 2002, pp. 234–245.

[12] N. Grech, J. Rathke, and B. Fischer, "JEqualityGen: Generating Equality and Hashing Methods," in *GPCE*, 2010, pp. 177–186.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," *SIGPLAN Not.*, vol. 37, pp. 58–70, 2002.

[14] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, pp. 92–106, 2004.

[15] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[16] W. L. Johnson and E. Soloway, "PROUST: Knowledge-Based Program Understanding," *IEEE Trans. Softw. Eng.*, vol. 11, no. 3, pp. 267–275, 1985.

[17] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *TACAS*, 2003, pp. 553–568.

[18] S. Khurshid and Y. L. Suen, "Generalizing symbolic execution to library classes," in *PASTE*, 2005, pp. 103–110.

[19] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[20] D. Marinov and S. Khurshid, "Valloy - virtual functions meet a relational language," in *FME '02*, 2002, pp. 234–251.

[21] 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*, 2009, pp. 342–352.

[22] C. R. Rupakheti and D. Hou, "An Empirical Study of the Design and Implementation of Object Equality in Java," in *CASCON '08*, 2008, pp. 111–125.

[23] ——, "An Abstraction-Oriented, Path-Based Approach for Analyzing Object Equality in Java," in *WCRE '10*, 2010, pp. 205–214.

[24] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *ASE*, 2010, pp. 43–52.

[25] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," in *CASCON '99*, 1999, pp. 125–135.

[26] M. Vaziri, F. Tip, S. Fink, and J. Dolby, "Declarative Object Identity Using Relation Types," in *ECOOP'07*, 2007, pp. 54–78.

[27] M. Vaziri-Farahani, "Finding bugs in software with a constraint solver," Ph.D. dissertation, MIT, MA, USA, 2004.