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
{rupakhcr, 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. We present 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 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 [1].

The issue of object equality is a general, long-standing design problem that can be traced back to Lisp [1]. 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 [2]–[5]. 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 [6], 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. We use the intra-procedural control flow graphs of Soot (Jimple) [7] 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 [8]. 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 the 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.
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.

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html. All URLs verified on June 23, 2010.

[2] http://www.eclipse.org/eclipse/

[3] http://www.netbeans.org/

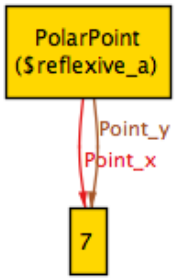


Fig. 1: A counter example for a reflexivity violation in the equality of the Point type hierarchy.

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 [9]. Furthermore, *PolarPoint* extends *Point* to provide the r and θ coordinates (Listing 2) in addition to x and y, and reuses the internal representations of *Point*.

Listing 1: A Point class in cartesian coordinate.

```
1  public class Point {
2   protected int x, y;
3   public Point(int x, int y) {
4    this.x = x;
5    this.y = y;
6   }
7   // ... getX() and getY() defined here
8   public boolean equals(Object o) {
9    if(o == null) return false;
10   if(o.getClass() != Point.class)
11    return false;
12   Point that = (Point)o;
13   return this.x == that.x && this.y == that.y;
14  }
15 }
```

Listing 2: A PolarPoint class in polar coordinate.

```
public class PolarPoint extends Point {
  public PolarPoint(int x, int y) {super(x, y);}
  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.

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

```
abstract sig Object {}  // Signature Decl.
sig Point extends Object {
   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* [7], and *Alloy* [8].

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 [7] is used to produce the inter-procedural paths. Specifically, CHA (Class Hierarchy Analysis), enhanced with more precise type information through data flow analysis, is used to resolve virtual method calls. Our checker prunes infeasible paths on the fly using path node-specific facts computed with

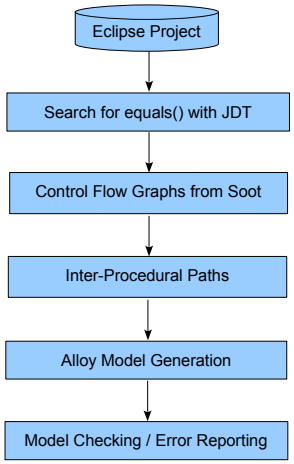
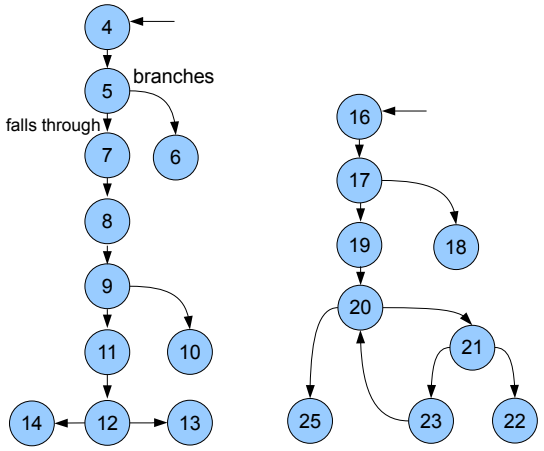


Fig. 2: The checker architecture.



(a) Person.equals() method (b) Person.eq() method

Fig. 3: Flow graphs for two methods of Person.

data flow analysis. The final paths are first checked for low level errors such as null pointer exceptions. They are then passed to an Alloy code generator for model generation. The Alloy models are then checked by the Alloy analyzer for violations of the equivalence relation. Any error discovered in the process is reported to the user through the Eclipse GUI.

III. Inter-procedural Path Generation

Inter-procedural path generation for an *equals* method starts with its control flow graph produced by Soot. Using this flow graph, a set of intra-procedural paths is first produced without expanding any call sites. These call sites are then selectively expanded and inlined to produce a new set of paths. This process is repeated until all the call sites have been appropriately expanded and inlined. A resulting path is then checked for feasibility, and any infeasible path is pruned. The final paths that survive the filtering process are used for further program analysis. Our checker also preserves the full context for each method call. The context-sensitivity is provided through cloning. Specifically, when a method is called multiple times, the path nodes from the method are cloned for each call separately.

The algorithm (Algorithm 1, which will be referred to as *main algorithm* here after) takes an entry method, m_e, as input (*equals* in this case) and returns a set of paths, S as output (Each path is a list of *PathNode*, a Jimple statement decorated with a context object). In the following subsections, we will describe each component of the algorithm using the *Person* class shown in Listing 4 as a running example.

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.

```
1  public class Person implements Cloneable {
2   private char[] name;
3   private char[] address;
4   public boolean equals(Object o) {
5    if(o==null || this.getClass() != o.getClass())
6     return false;
7    Person that = (Person)o;
8    boolean t = this.eq(this.name, that.name);
9    if(t != true)
10    return false;
11   t = that.eq(that.address, this.address);
12   if(t != true)
13    return false;
14   return true;
15  }
16  protected boolean eq(char[] a, char[] b) {
17   if(a.length != b.length)
18    return false;
19   int i = 0;
20   while(i < a.length) {
21    if(a[i] != b[i])
22     return false;
23    ++i;
24   }
25   return true;
26  }
27 }
```

B. IntraProcedurePathGenerator

The *IntraProcedurePathGenerator* algorithm (Algorithm 2) takes a flow graph $G = (V, E)$ as input and returns a set of intra-procedrual 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.

Algorithm 1 Inter-procedural path generation algorithm.

Require: m_e, an entry method
Require: T_e, the receiver type
Require: Q, an empty queue of paths
Require: S, an empty set of paths
1: $G_e \leftarrow FlowGraph(m_e)$ {Soot's ExceptionalUnitGraph}
2: $P_e \leftarrow IntraProcedurePathGenerator(G_e)$
3: {Associate null context to each path.}
4: $P_\emptyset \leftarrow Context(P_e, [null, m_e, T_e])$
5: $F_\emptyset \leftarrow Filter(P_\emptyset)$ {Filter infeasible paths}
6: **for all** $p_\emptyset \in F_\emptyset$ **do**
7: $Enqueue(p_\emptyset, Q)$
8: **end for**
9: **while** $Length(Q) \neq 0$ **do**
10: $p \leftarrow Dequeue(Q)$
11: $c \leftarrow CallSite(p)$ {First un-processed call site in p}
12: **if** $c = null$ **then**
13: add p to S {No call sites to be processed}
14: **else**
15: **if** $Expandable(c, p)$ **then**
16: $M \leftarrow CHA(c, p)$ {Returns method and type pairs}
17: **for all** pair $[m, T] \in M$ **do**
18: $G_m \leftarrow FlowGraph(m)$
19: $P_m \leftarrow IntraProcedurePathGenerator(G_m)$
20: $P_{c_+} \leftarrow Context(P_m, [c, m, T])$
21: **for all** path $p_{c_+} \in P_{c_+}$ **do**
22: $t \leftarrow Clone(p)$
23: $n \leftarrow Inline(p_{c_+}, t)$ {Inline p_{c_+} into t}
24: $F \leftarrow Filter(\{n\})$
25: **for all** path $f \in F$ **do**
26: $Enqueue(f, Q)$
27: **end for**
28: **end for**
29: **end for**
30: **end if**
31: **end if**
32: **end while**
33: **return** S

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 Lisiting 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().

```
1 4  5  6
2 4  5  7  8  9  10
3 4  5  7  8  9  11  12  13
4 4  5  7  8  9  11  12  14*
```

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).

Algorithm 2 Intra-procedural path generation algorithm.

Require: $G = (V, E)$, a directed flow graph
Require: Q, an empty queue of paths
Require: S, an empty set of paths
r $\leftarrow$ root(G)
$Enqueue([r], Q)$ {Adds first path to the queue}
while $Length(Q) \neq 0$ **do**
$p \leftarrow Dequeue(Q)$ {Polls a path to work with}
$l \leftarrow LastNode(p)$ {Gets last node of path p}
$C \leftarrow Children(l)$ {Gets list of successors of $l \in G$}
if $Length(C) = 0$ **then**
add p to S
else
for $i = 0 \rightarrow Length(C) - 1$ **do**
$c \leftarrow Count(C[i], p)$ {Counts all $C[i] \in p$}
{Ensures a loop is entered only once in a path}
if $c \leq 1$ **then**
$p' \leftarrow p + C[i]$ {p is unchanged}
$Enqueue(p', Q)$
end if
end for
end if
end while
return S

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

```
1 16  17  18
2 16  17  19  20  25*
3 16  17  19  20  21  22
4 16  17  19  20  21  23  20  25*
5 16  17  19  20  21  23  20  21  22
```

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 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. In 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():* Method call on a field is not expanded to keep the implementation simple. This sometimes helped us 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 special 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 [10]. 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 dequeued 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, 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 Lisiting 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.

```
1 4 5 7 8 16 17 18 8 9 11 12 14
2 4 5 7 8 16 17 19 20 25 8 9 11 12 14*
3 4 5 7 8 16 17 19 20 21 22 8 9 11 12 14
4 4 5 7 8 16 17 19 20 21 23 20 25 8 9 11 12 14*
5 4 5 7 8 16 17 19 20 21 23 20 21 22 8 9 11 12 14
```

As an example of an infeasible path, consider path 1 (Lisiting 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 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.

The paths that pass the filter, paths 2 and 4, are added to the queue for further processing (line 26 of the main algorithm). The algorithm further processes the *that.eq()* calls at node 11 for these two paths. Listing 8 shows the final paths.

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

```
1 4 5 7 8 16 17 19 20 25 8 9 11 16 17 19 20 25
  11 12 14
2 4 5 7 8 16 17 19 20 25 8 9 11 16 17 19 20 21
  23 20 25 11 12 14
3 4 5 7 8 16 17 19 20 21 23 20 25 8 9 11 16 17
  19 20 25 11 12 14
4 4 5 7 8 16 17 19 20 21 23 20 25 8 9 11 16 17
  19 20 21 23 20 25 11 12 14
```

IV. Alloy Model Generation

The goal of this step in our checker is to recognize the high-level, predefined abstractions that can be used to define equality from the low-level, inter-procedural paths generated for an *equals* in Java. Specifically, code pattern detectors are used to query the structure as well as the execution order of the statements in the paths to detect these patterns/abstractions. These abstractions are collected to form an Alloy model.

An overview of the Alloy model is provided in Section IV-A. To reliably detect abstractions from code patterns, our checker also uses information about data flow between variables and reference aliasing as well as the information about the structure of the object graph. We handle the former with what we call *copy root analysis* (Section IV-B) and the latter with *object root analysis* (Section IV-C). Sections IV-D, IV-E, and IV-F describe the various equality-related patterns and how they are modeled in Alloy.

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* tranlation 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 substyping. 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 [8].

The *Devleoper* 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.

```
1  abstract sig Object {}
2
3  sig Cloneable in Object {}
4
5  sig Person extends Object {
6   name : seq Int,
7   address : seq Int
8  }
9  fact { Person in Cloneable }
10
11 sig Developer extends Person {}
12
13 // Equals predicate
14 pred Object :: equals( that: Object ) {
15  (this in Person - Developer) => // Person
16  (
17   (that in Person - Developer) and
18   (this.name = that.name) and
19   (this.address = that.address)
20  )
21  else
22  (this in Developer) =>      // Developer
23  (
24   (that in Developer) and
25   (this.name = that.name) and
26   (this.address = that.address)
27  )
28 }
29 // Equivalence relation specification. Elided.
```

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 following rules:

- *CopyRoot(Var) = Expression*: This rule applies if the path has a definition statement *Var = Expression* that is not killed before the evaluation point. For example, for path *[x = a-b; return x;]*, *CopyRoot(x) =a-b* after *x = a-b;*.
- *CopyRoot(Var) = CopyRoot(Var1)*: If the path has a definition statement *Var = Var1* that is not killed before the evaluation point. For example, for path *[x = this; y = x; return y;]*, *CopyRoot(y) = this* after *y = x;*.
- *CopyRoot(Formal Parameter) = Formal Parameter* for entry method.
- *CopyRoot(Formal Parameter) = 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:

- *ObjectRoot(this) = this*: Object root of *this* is itself.
- *ObjectRoot(r.f) = ObjectRoot(r)*: e.g., ObjectRoot(this.field) = ObjectRoot(this) = this.
- *ObjectRoot(a[e]) = ObjectRoot(a)*: The object root of an array access is the object root of the array.
- *ObjectRoot(Var) = ObjectRoot(Expression)*: This rule applies if the path has a definition statement *Var = Expression* that is not killed before the evaluation point. For example, for path *[a = this; b = a; c = b.field; d = c; return d;]*, after *d = c;*, it holds *ObjectRoot(d) = ObjectRoot(c) = ObjectRoot(b.field) = ObjectRoot(b) = ObjectRoot(a) = ObjectRoot(this) = this*.
- *ObjectRoot(Formal Parameter) = Formal Parameter* for entry method.
- *ObjectRoot(Formal Parameter) = ObjectRoot(Actual Parameter)* for non entry method.
- *ObjectRoot(Others) = 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.

```
1  boolean equals(Object o) {
2   boolean t = o instanceof Circle;
3   if(t != true)
4    return false;
5   Circle that = (Circle)o;
6   int thisR = this.radius;
7   int thatR = that.radius;
8   if(thisR != thatR)
9    return false;
10  return true;
11 }
```

D. Type Checking Pattern

To illustrate how *CopyRoot* is used in detecting type checking patterns, consider the *equals* method for the *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 *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 *CopyRoot(t)* is an *instanceof* expression or not, which is true. So it further checks if an operand of the *instanceof* expression is the parameter of *equals*. This is also true (CopyRoot(o) == o). Thus, it concludes that line 3 performs an *instanceof* type checking and abstracts it as *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)o; ...* } *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:

- $this[\{.f\}|\{.m()\}]^+ == that[\{.f\}|\{.m()\}]^+$, or $this[\{.f\}|\{.m()\}]^+.equals(that[\{.f\}|\{.m()\}]^+)$. 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.
- $this[\{.f\}|\{.m()\}]^+.compareTo(that[\{.f\}|\{.m()\}]^+) == 0$ 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*.
- $SM(this[\{.f\}|\{.m()\}]^+) == SM(that[\{.f\}|\{.m()\}]^+)$. For example, *Math.atan(this.f) == Math.atan(that.f)* is translated to *this.Math_atan_f = that.Math_atan_f* in Alloy.

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 speficied 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.

Description	Tomcat 6	Lucene 3.0	JDK 1.5
Interfaces	175	76	1745
Classes	1237	888	11240
Type hierarchies	28	67	484
Defined equals	29	96	623
Processed equals	31	98	694

TABLE I: Characteristics of the inspected projects.

V. Evaluation Results

The equals checker has been tested with three medium to large, open source, real world projects: Tomcat 6 [4], Lucene 3.0 [5], 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 sever 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.

A. Path Generation and Filtering

Table II summarizes path generation for the three projects. The *total paths* constitute the filtered paths, the intermediate

[4]http://tomcat.apache.org/
[5]http://lucene.apache.org/

Description	Tomcat 6	Lucene 3.0	JDK 1.5
Total paths	515	1180	44788
Boolean path filtering	393	727	27925
Nullness analysis filtering	0	0	845
Type analysis filtering	0	0	125
Throw analysis filtering	0	0	3043
Other intermediate paths	45	132	8259
Final working paths	**77**	**321**	**4591**
Path reduction	**85.04%**	**72.79%**	**89.74%**

TABLE II: Summary of path generation and filtering.

Description	Tomcat 6	Lucene 3.0	JDK 1.5
Type checking	26	103	794
State comparison	48	314	2165
Array comparison	1	0	33
List comparison	0	0	19
Set comparison	0	0	2
Map comparison	1	0	31
Total detected abstractions	**76**	**417**	**3044**
Unhandled equals	**0**	**6**	**80**

TABLE III: Summary of detected abstractions.

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 stategies reduce paths from 72.79% to 89.74%. Even for a large project like JDK 1.5, we need to process merely a few thousand paths (4591) for the Alloy model generation. 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* implementations, we found that about half 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. Although the three projects are mature and rigorously tested industrial projects, the checker still identified several violations. Therefore, it would be reasonable to assume that such a checker would be more useful during routine development. Note that the numbers presented in Table IV represent faulty type hierarchies. We want to emphasize that in general, 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.

Description	Tomcat 6		Lucene 3.0		JDK 1.5	
	Prob.	False P.	Prob.	False P.	Prob.	False P.
Reflexivity	0	0	0	0	3	2
Symmetry	1	0	1	0	10	0
Transitivity	1	0	0	0	8	3
Total	**2**	**0**	**1**	**0**	**21**	**5**

TABLE IV: Summary of detected problems.

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

TABLE V: Summary of execution time.

D. Execution Time

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

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

VI. Related Work

A. Equals Design and Implementations

Various authors, including us [4], have investivated design guidelines for equality [2], [6], [11].Vaziri et al. [3] extend Java with relation types with which equality can be specified in terms of object properties and *equals* implementation can be generated automatically. 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 [5]. They use Java Collection Framework as one of their benchmarks where they modify existing source code to add relevant annotations. 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* [12].

Saturn is an intra-procedurally path-sensitive and summary based, context-sensitive program analysis framework for C [13], [14]. 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 [15] 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 [16]. 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

The idea of abstraction recognition has been investigated previously. For example, Johnson and Soloway have developed the PROUST tool for interactive analysis and understanding of Pascal programs to support novice programmers [17]. Seemann and Gudenberg propose a technique for detecting design patterns [18]. Their approach of relying on some well-defined names for abstraction is similar to us. 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. Balmas describes the query by outlines tool called QBO [19]. QBO can answer queries expressed as constraints on an outline model. 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 [8]. However, equivalence relation in particular is not discussed. Marinov et al. present a way to extend Alloy so that it can model virtual functions more naturally [20].

D. Existing Checkers

There are several static checkers available for Java. FindBugs [6] 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. [21], [22] use SCL to specify and detect violation of implementation constraints for *equals()* within each individual class. 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.

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

References

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

[2] J. Bloch, *Effective Java programming language guide*. Addison-Wesley, 2001.

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

[4] 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.

[5] 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.

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

[7] 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.

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

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

[10] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP '95*, 1995, pp. 77–101.

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

[12] B. G. Ryder, "Dimensions of precision in reference analysis of object-oriented programming languages," in *CC'03*, 2003, pp. 126–137.

[13] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the saturn project," in *PASTE '07*, 2007, pp. 43–48.

[14] I. Dillig, T. Dillig, and A. Aiken, "Sound, complete and scalable path-sensitive analysis," in *PLDI'08*, 2008, pp. 270–280.

[15] M. Das, S. Lerner, and M. Seigle, "Esp: path-sensitive program verification in polynomial time," in *PLDI '02*, 2002, pp. 57–68.

[16] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, "Path invariants," in *PLDI '07*, 2007, pp. 300–309.

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

[18] J. Seemann and J. W. von Gudenberg, "Pattern-based design recovery of java software," in *SIGSOFT '98/FSE-6*, 1998, pp. 10–16.

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

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

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

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