Constructing Precise Object Relation Diagrams

Ana Milanova Atanas Rountev Barbara G. Ryder
Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
{milanova,rountev,ryder}@cs.rutgers.edu

Abstract

The Object Relation Diagram (ORD) *of a program is a class interdependence diagram which has applications in a wide variety of software engineering problems (e.g., integration testing, integration coverage analysis, regression testing, impact analysis, program understanding, and reverse engineering). Because the imprecision of the ORD directly affects the practicality of its usage, it is important to investigate techniques for constructing precise ORDs.*

This paper makes three contributions. First, we develop the Extended Object Relation Diagram (ExtORD)*, a version of the ORD designed for use in integration coverage analysis. The ExtORD shows the specific statement that creates an interclass dependence, and can be easily constructed by extending techniques for ORD construction. Second, we develop a general algorithm for ORD construction, parameterized by class analysis. Third, we demonstrate empirically that relatively precise class analyses can significantly improve diagram precision compared to earlier work, resulting in average size reduction of 55% for the ORD and 39% for the ExtORD.*

1 Introduction

Object-oriented systems are characterized by complex interclass interactions. The goal of *integration testing* is to reveal faults that cause some interclass interactions to fail. One fundamental problem in integration testing is how to define the test order (i.e., should class A be tested before or after class B is tested). The goal of *integration coverage analysis* is to show that sufficiently many interclass interactions are covered during testing. Coverage analyzers need to determine which statements trigger interclass dependences and which interclass dependences are triggered at a given statement (i.e., exactly which two classes are involved). The goal of *regression testing* is to show that after a change is made, the program still satisfies its requirements. Regression testing must address the following problems: (i) determining the impact of a change on a given class (i.e., the set of classes affected by the change) and (ii) defining the regression test order (i.e., the order in which affected classes need to be retested).

The *Object Relation Diagram (ORD)* [13] is a model of interclass dependences which can be used to address these problems. The ORD can be used to define efficient test order for integration testing. It can be used in coverage analysis to determine pairs of interacting classes. In addition, the ORD can be used in regression testing to find the set of classes affected by a change and to define efficient regression test order.

The ORD of a program P is a directed graph in which nodes represent program classes and edges represent dependences between these classes. There are three kinds of edges. An *inheritance edge* from B to A represents that B depends on A because B is a subclass of A. An *aggregation edge* from B to A represents that B depends on A because instances of class A may be contained by instances of class B. An *association edge* from B to A represents associations between objects of class B and objects of class A due to method calls or field accesses. (Several diagrams are shown in Figure 2 in Section 2.)

One disadvantage of the ORD is that there is at most one edge of each kind between two classes. Therefore, coverage analyzers cannot use the ORD to determine which specific statement triggers the dependence. We define the *Extended ORD (ExtORD)* to address this problem. The ExtORD allows multiple edges of each kind, one for each statement that triggers this kind of dependence; therefore, test coverage of such statements can be distinguished.

Imprecise ORDs and ExtORDs contain spurious dependence edges, representing interclass dependences that are impossible and do not correspond to any actual program execution. In integration testing, spurious edges may lead to dependence cycles in the ORD, which complicates the task of defining a test order. In integration coverage analysis, spurious dependence edges result in time spent on trying to execute code in a way which triggers impossible dependences. In regression testing imprecision leads to two problems: (i) when the set of classes determined to be affected by a change is too imprecise, time will be wasted on retesting unaffected classes and (ii) cycles in the ORD complicate the definition of a regression test order. In these cases, as well as for other ORD uses such as program understanding and reverse engineering, significant time and effort could be saved if the ORD and the ExtORD are more precise (i.e., contain fewer spurious edges).

Because of the wide range of applications of these dependence diagrams, it is important to investigate approaches for construction of precise ORDs and ExtORDs. In order to construct these diagrams, it is necessary to have information about the classes of all objects that certain variables may refer to at run time. This information can be obtained by using *class analysis*, which is a popular form of static program analysis for object-oriented languages. The goal of class analysis is to compute for each variable r the set $Cs(r)$ of all classes such that an object of class $C \in Cs(r)$ may be bound to r at run time. There are many class analyses with different tradeoffs between cost and precision, developed primarily in the context of optimizing compilers for object-oriented programming languages. The precision of the dependence diagrams directly depends on the precision of the underlying class analysis which is used during ORD/ExtORD construction.

In this paper we define a generalized algorithm for ORD construction which is parameterized by class analysis. By varying the underlying class analysis, the algorithm allows the user to build ORDs with differing degrees of precision. For presentation purposes we use Java programs to demonstrate our approach, but the methodology can be used with minor modifications with other object-oriented programming languages.

Previous work uses the structure of the program class hierarchy to determine $Cs(r)$ in order to construct the ORD. In our experiments, we compare this simple form of class analysis with two more precise class analyses based on algorithms presented in [20] and [16]. On a set of nine realistic Java programs, our results show that these two more precise analyses significantly improve the precision of the ORD and the ExtORD, compared to using the structure of the class hierarchy. We observed average reductions of 55% for the number of edges in the ORD and 39% for the number of edges in the ExtORD.

In integration testing, these reductions may result in substantial savings in time and effort. When using coverage analysis tools, substantially less time will be spent trying to exercise impossible interclass dependences. In regression testing, the improved precision leads to (i) smaller number of classes selected for retesting and (ii) less time spent on determining a retest order. Furthermore, the significantly improved precision is beneficial for other uses of the ORD/ExtORD (e.g., for program understanding and reengineering).

Even though most of our data programs are large, the experimental results show that the two precise class analyses from [20, 16] have practical cost. This practicality makes them realistic candidates for use in software engineering tools.

Contributions The contributions of our work are the following:

- We define the Extended Object Relation Diagram (ExtORD) which is suitable for integration coverage analysis. Any method used to construct the ORD from program code can be trivially extended to construct the ExtORD.
- We define a generalized algorithm for ORD construction which is parameterized by a class analysis. The parameterization allows the user to control the cost/precision tradeoff by varying the class analysis.
- We demonstrate empirically that two practical class analyses [20, 16] improve substantially the precision of the ORD/ExtORD, compared to using the structure of the class hierarchy. Our results show that employing these analyses in ORD construction tools may result in significant savings in time and effort.

Outline The rest of this paper is organized as follows. Section 2 describes applications of the ORD, summarizes previous work on ORD construction, and argues the importance of ORD precision. Section 3 presents a general algorithm for ORD construction and discusses specific class analyses. The experimental results are presented in Section 4. Section 5 discusses related work and Section 6 presents conclusions and future work.

2 The Object Relation Diagram

2.1 Applications of Object Relation Diagrams

The ORD can be used in integration testing, integration coverage analysis, regression testing, impact analysis, program understanding, and reverse engineering. Once constructed, the ORD can be reused by various clients at no additional cost. In this section we briefly discuss several specific client applications of the ORD.

Integration Testing One goal of integration testing is to reveal faults that are triggered by the interactions between classes. Usually, integration testing proceeds in stages. At each stage there is a target set of classes under test. Classes not included in the target set and used by some of the classes in the target set need to be simulated by *stubs*. The stubs simulate the class behavior for the given context, which is usually a small subset of the entire class behavior. One important problem in integration testing is how to determine the order in which classes are tested (i.e., should class A be tested before or after class B is tested).

Bottom-up integration testing strategies [5, 13, 14] aim at minimizing the number of stubs, because stub construction requires significant time and effort. In this case, the test order can be derived by bottom-up traversal of the ORD. Clearly, no stub is required for a class that is tested before the classes that depend on it. Intuitively, the independent classes are tested first, then the classes that depend on them, and so on.

Coverage Analysis The goal of coverage analysis is to evaluate the quality of a given test suite by measuring the coverage of program code and of certain aspects of the behavior of that code. Integration testing focuses on faults due to complex interclass dependences. Thus, it is important to ensure that the test suite covers interclass interactions. For example, one coverage requirement is that all interclass method calls (i.e., calls from one class to a method in another class) should be exercised [5]. Another more strict requirement states that every possible interclass dependence should be covered [24]. The Extended ORD (described in Section 2.2) is a version of the ORD that can be used by coverage analyzers to determine the set of statements that trigger interclass dependencies and therefore need to be exercised in order to satisfy coverage requirements.

Regression Testing The goal of regression testing is to ensure that when a change is made to a program, the program still satisfies its requirements. Because of complex dependences between classes, when a change is made to a class, this change usually affects other classes in the program. Each of these affected classes needs to be retested. Two fundamental problems in regression testing are (i) how to identify the classes affected by a change and (ii) how to efficiently perform retesting of these affected classes. The ORD can be used in regression testing to identify the set of affected classes. All classes reachable backwards in the ORD from the changed classes are potentially affected by the change. In addition, the ORD can be used to determine a test order which minimizes the necessary stubs. Such test order leads to an efficient retesting strategy, because stub construction requires significant effort. Similarly to determining the test order for bottom-up integration testing, the regression test order can be derived by bottom-up traversal of the ORD (i.e., class A is retested before the classes that depend on it; when the classes that depend on A are retested, stubs are not required for A because it has already been retested).

```
   class X { void n() {...} }
   class Y extends X { void n() {...} }
   class Z extends X { void n() {...} }

   class A {
       X f;
2      A(X xa) { this.f = xa; ...  }
       void m() {
3          X xa = this.f;
4          xa.n(); } }

   class B {
       X g;
5      B(X xb) { this.g = xb; ...  }
       void m() {
6          X xb = this.g;
7          xb.n(); } }

8  s1: Y y = new Y();
9  s2: Z z = new Z();
10 s3: A a = new A(y);
11 s4: B b = new B(z);
12     a.m();
13     b.m();
```

Figure 1. Sample set of statements.

2.2 The ORD by Kung et al.

Kung et al. [13] define the Object Relation Diagram using three kinds of dependence edges. For the rest of the paper we use the notation $(\langle B, l\rangle, A)$ to denote an edge labeled l from B to A.

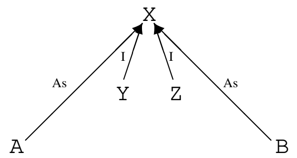
Inheritance There is an edge labeled I from class B to class A if and only if B is a direct subclass of A.

Aggregation There is an edge $(\langle B, Ag\rangle, A)$ if and only if statement `this.f = new A(..)` appears in a constructor defined in class B.[1]

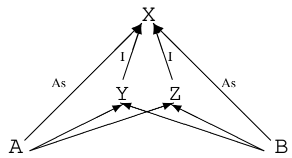
Association There is an edge $(\langle B, As\rangle, A)$ if and only if one of the following is true:

- `T m(...,A r,...) {...}` is a definition of a method or a constructor in class B. We refer to such an association as a *parameter association*.
- Field access `r.f` appears in a statement contained by a method or a constructor defined in class B and the declared type of reference variable r is A ($r \neq$ `this`). We refer to such an association as a *field access association*.
- Method invocation `r.m(...)` appears in a statement contained by a method or a constructor in class B and the declared type of r is A ($r \neq$ `this`). We refer to such an association as a *method call association*.

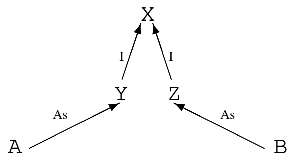
[1] According to [13] aggregation can be (i) static, due to encapsulated non-pointer fields and (ii) dynamic, due to pointer fields initialized within constructors. Since all fields in Java are references (i.e., pointers to objects), static aggregation is not possible and we simplify the definition accordingly.



(a) Static dependences



(b) Transitive subclasses



(c) Precise run-time dependences

Figure 2. Object Relation Diagrams corresponding to different methods of construction.

Figure 2(a) shows the ORD for the program in Figure 1. Clearly, this diagram does not reflect dependences due to dynamic bindings of polymorphic variables. For example, class A depends on class Y because at run time an object of class Y is bound to polymorphic variable `xa` at line 4. However, this dependence is not shown in the ORD from Figure 2(a).

The *class firewall for class* C, denoted by $CFW(C)$, is defined as the set of classes reachable from the node corresponding to class C in the transpose of the ORD [13, 14]. Intuitively, the class firewall contains the classes that may be affected when C is modified, and thus should be retested (assuming that the ORD for the program is not modified). For example, based on the ORD in Figure 2(a), $CFW(X) = \{X, Y, Z, A, B\}$ and $CFW(Y) = \{Y\}$.

In this paper we define the *Extended Object Relation Diagram (ExtORD)* which makes the dependence diagram more informative and more suitable for use in code coverage analysis tools. The ExtORD contains *annotated* association and aggregation edges. There is an annotated aggregation $Ag : s_i$ if and only if s_i is an object creation statement which triggers an aggregation. There is an annotated association $As : s_i$ if and only if s_i is a program statement which triggers a field access association or a method call association. There is a non-annotated association As which denotes parameter association. Thus, the ExtORD may contain multiple aggregation and multiple association edges between two nodes, because it makes explicit which program statements trigger the dependences. For example, association edge $(\langle A, As : 4\rangle, X)$ indicates that because of the virtual call at line 4 in Figure 1, there is an association between class A and class X. It is easy to see how coverage analyzers can make use of the Extended Object Relation Diagram. Since the ExtORD identifies the interclass dependences together with the statements that create them, a high-quality test suite should exercise these statements and should achieve sufficient coverage of the dependencies triggered by the statements.

2.3 The ORD by Labiche et al.

Recall from Section 2.2 that the ORD from [13] does not reflect the dependences that arise due to possible dynamic bindings of polymorphic variables. In order to correct this problem, Labiche et al. [14] propose to augment the ORD from [13] with additional edges representing dynamic relationships. If there is an edge labeled As from A to B, there are additional edges with the same label from A to all subclasses of B (including transitive subclasses). For the example in Figure 1, there are additional association edges from A and B to both Y and Z. Figure 2(b) shows how the diagram in Figure 2(a) is augmented with these additional edges. This approach is equivalent to modifying the three rules for association inference from Section 2.2 to take into account the possible dynamic bindings of polymorphic variables by considering the structure of the class hierarchy (i.e., by considering all transitive subclasses of the targets of ORD association edges).

The ExtORD can be augmented in a similar fashion. If there is an edge labeled $As : s_i$ from A to B, there are edges labeled $As : s_i$ from A to all subclasses of B (including transitive subclasses). For the set of statements in Figure 1, there are additional edges $(\langle A, As : 4\rangle, Y)$ and $(\langle A, As{:}4\rangle, Z)$. In order to achieve good coverage of interclass dependences, it may not be enough to exercise a statement once. Due to polymorphism, certain statements (e.g., the method call at line 4 in Figure 1) need to be exercised several times to achieve coverage for all dynamic interclass dependences that may result from such statements.

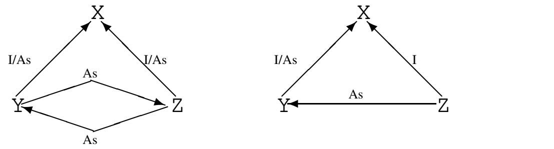
2.4 The Disadvantages of Imprecise Analysis

Clearly, the ORD computed by Kung et al. [13] omits dependences and may result in incomplete testing and retesting. On the other hand, the ORD computed by considering the structure of the class hierarchy could be overly conservative. This imprecision results from imprecise analysis of the possible dynamic bindings of polymorphic variables. In Figure 1 class A does not depend on class Z because at run time `A.A.xa` at line 2 and `A.m.xa` at line 4 can reference only objects of class Y. The same kind of spurious dependence occurs between class B and class Y. The exact ORD corresponding to this set of statements is shown in Figure 2(c).

Analysis imprecision results in spurious dependences and can impair the usefulness of tools that employ the ORD

```
  class X { void n() {...} }
  class Y extends X {
1     Y(X xy) { xy.n(); ...  } ...  }
  class Z extends X {
2     Z(X xz) { xz.n(); ...  } ...  }
3  s1: X x = new X();
4  s2: Y y = new Y(x);
5  s3: Z z = new Z(y);
```

Figure 3. Sample program.



(a) Transitive subclasses (b) Precise run-time dependences

Figure 4. ORDs corresponding to Figure 3.

or the ExtORD. When the ORD is traversed bottom-up to choose integration test order or regression test order, imprecision can lead to *dependence cycles*. There are two problems with dependence cycles: (i) complex analysis and additional work are required to break the cycle [12, 24] and (ii) cycle breaking requires stub construction. Consider the ORDs in Figure 4 which correspond to the set of statements in Figure 3. The diagram in Figure 4(a) is constructed based on the structure of the class hierarchy, and contains a dependence cycle between Y and Z.[2] This cycle is caused by a spurious dependence edge from Y to Z which is due to the imprecise analysis of the possible dynamic bindings of `xy`—according to the class hierarchy, `xy` may refer to instances of X, Y, or Z. Therefore, the test order cannot be determined without cycle breaking. In contrast, the precise diagram in Figure 4(b) clearly shows that the test order should be X, Y, Z.

In coverage analysis, imprecision can lead to incorrect reports of inadequate coverage. For example, if the structure of the class hierarchy is used to compute the ExtORD for the statements in Figure 1, the following three association edges would correspond to the method call at line 4:

$$(\langle A, As\!:\!4\rangle, X) \quad (\langle A, As\!:\!4\rangle, Y) \quad (\langle A, As\!:\!4\rangle, Z)$$

A coverage analyzer will always report that the first and the third edge are not covered. Therefore, the user will attempt to exercise the call with receivers from classes X and Z and will spend valuable time determining that variable `xa` cannot refer to objects of any other class except Y.

[2] In Figure 4(a) there are self-loop association edges at Y and Z; for simplicity, we omit them from the picture. Clearly, self-loop edges can be ignored when the ORD is used for test order definition.

In regression testing, an imprecise ORD can lead to large class firewalls, and thus code not affected by the change may be selected for retesting.

Imprecision in the ORD results in waste of time and effort for activities such as breaking of spurious dependence cycles, trying to execute a statement in a manner that triggers nonexistent interclass dependences, and retesting parts of the code that are not affected by a change. Therefore, it is important to investigate approaches for constructing more precise ORDs and ExtORDs.

3 Class Analysis for ORD Construction

Recall that information about the dynamic bindings of polymorphic variables is necessary during ORD construction. In order to determine all classes for which field access `p.f` triggers field associations with the enclosing class, one needs to find the possible dynamic bindings of reference variable `p` (i.e., the classes of all objects that `p` may refer to at run time). Similarly, for method call associations one needs the set of all possible classes of the receiver object; for parameter associations one needs the set of all classes of objects that can be bound to formals.

This information can be obtained by using *class analysis*, which is a popular form of static program analysis originally developed in the context of optimizing compilers. Class analysis computes a set of classes for each program variable r; this set approximates the classes of all run-time objects that may be bound to r. There is a wide variety of existing class analyses with various degrees of cost and precision [17, 1, 2, 18, 9, 4, 11, 8, 6, 19, 21, 22, 23, 25, 15, 20, 10, 16]. These analyses can be used for ORD and ExtORD construction. More precise underlying class analyses result in fewer spurious dependences and therefore more precise ORDs and ExtORDs. For the set of statements in Figure 3, the set of possible classes of all objects that may be bound to variable `xy` at line 1 is $\{X\}$. The set $\{X, Y, Z\}$ computed by examining the class hierarchy is a valid approximation because it includes the only possible run-time class X, but it is imprecise because it also includes Y and Z.

In Section 3.1 we present a general algorithm for ORD construction, parameterized by a class analysis. Section 3.2 discusses *Class Hierarchy Analysis* (*CHA*) which is the simplest form of class analysis [7]. Section 3.3 discusses two class analyses that are more precise than *CHA*.

3.1 ORD Construction

Figure 5 presents an algorithm, parameterized by a class analysis, which computes the ORD starting from an empty ORD. Cs is the output of the given class analysis. $Cs(x)$ denotes the set of classes which approximates the classes

input Stmts: set of statements
$\mathit{Methods}$: set of methods
Cs: $\mathit{Vars} \rightarrow \mathit{Powerset}(\mathit{Classes})$
output ORD

foreach direct subclass X of class C **do**
  $\mathit{ORD} := \mathit{ORD} \cup \{(\langle X, I\rangle, C)\}$
foreach s: $\mathit{this}.f = \mathit{new}\ C \in \mathit{Stmts}$ in a constructor **do**
  $\mathit{ORD} := \mathit{ORD} \cup \{(\langle \mathit{EnCl}(s), \mathit{Ag}\rangle, C)\}$
foreach $s \in \mathit{Stmts}$ containing $p.f$ **do**
  if $p \neq$ `this` **do**
    $\mathit{ORD} := \mathit{ORD} \cup \{(\langle \mathit{EnCl}(s), \mathit{As}\rangle, C) \mid C \in \mathit{Cs}(p)\}$
foreach virtual call $s: l = p.m(...) \in \mathit{Stmts}$ **do**
  if $p \neq$ `this` **do**
    $\mathit{ORD} := \mathit{ORD} \cup \{(\langle \mathit{EnCl}(s), \mathit{As}\rangle, C) \mid C \in \mathit{Cs}(p)\}$
foreach method $m \in \mathit{Methods}$ **do**
  foreach explicit formal parameter p of m **do**
    $\mathit{ORD} := \mathit{ORD} \cup \{(\langle \mathit{EnCl}(m), \mathit{As}\rangle, C) \mid C \in \mathit{Cs}(p)\}$

Figure 5. ORD construction.

```
  class Y {...}
  class Z extends Y {...}
  class X {
    Y f;
    void set(Y r)
1       { this.f = r; }
    static void main() {
2     s1: X p = new X();
3     s2: Y q = new Y();
4        p.set(q); } }
```

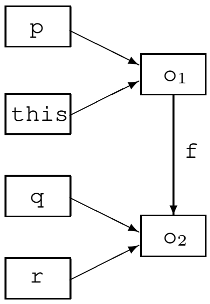


Figure 6. Sample program and its points-to graph.

of all objects that may be bound to variable x. EnCl denotes the enclosing class of a given statement or method. Figure 5 shows the process of constructing different ORD edges. For example, for each statement containing a virtual call through variable p, the algorithm adds association edges from the node representing the class enclosing the statement to each node representing a class $C \in \mathit{Cs}(p)$. For brevity we omit discussion of calls to static methods and accesses of static fields; our implementation handles these cases properly. Clearly, the ExtORD can be constructed by extending the algorithm from Figure 5 to add annotated edges.

3.2 Class Hierarchy Analysis

Class Hierarchy Analysis (CHA) is the simplest form of class analysis. To determine the possible bindings of polymorphic variables, CHA examines the structure of the class hierarchy. For a given variable r of declared type C, the set of possible classes of objects that may be bound to r is reported to be the set containing C and all direct and transitive subclasses of C (excluding abstract classes). For example, *CHA* computes the following sets $Cs(x)$ for the variables in Figure 6:

$$Cs(X.set.r) = \{Y, Z\} \quad Cs(p) = \{X\} \quad Cs(q) = \{Y, Z\}$$

3.3 Class Analysis Based on Points-to Analysis

Points-to analysis is a fundamental static analysis which determines the set of objects whose addresses may be stored in reference variables and reference object fields. These *points-to sets* are typically computed by constructing one or more *points-to graphs*, which serve as abstractions of the run-time memory states of the analyzed program. A sample program and its points-to graph are shown in Figure 6. The points-to graphs contain two kinds of edges, both of which are presented in Figure 6. For example, edge (p, o_1) shows that reference variable p may point to object o_1, where object name o_1 represents all objects that may be created at object allocation site s_1. Edge $(\langle o_1, f\rangle, o_2)$ shows that field f of object o_1 may point to object o_2.

The points-to solution can be used to derive the solution of a corresponding class analysis. The set of possible classes of objects that may be bound to r can be determined by examining the classes of all objects in the points-to set of r.

In this section we briefly present at a high level two points-to analyses, denoted by *AND-PT* and *OBJ-PT*, and their corresponding class analyses, denoted by *AND* and *OBJ*. *AND-PT* [20] is a flow-insensitive and context-insensitive points-to analysis and *OBJ-PT* [16] is a context-sensitive version of *AND-PT*.[3] We do not intend to present all details of the design and implementation of these analyses; more detailed information is available in [20, 16].

AND-PT is derived from Andersen's points-to analysis for C [3]. The analysis uses *rules* that add new edges to points-to graphs. Each rule represents the meaning of a program statement. For example, statement $l = r$ creates new points-to edges from l to all objects pointed to by r. Similarly, statement $l.f = r$ creates new field edges labeled f from all objects in the points-to set of l to all objects in the points-to set of r. For virtual call statement $l = r_0.m(r_1, ..., r_k)$, the target method is determined based on the receiver objects pointed to by r_0 and the method identifier m. The rules are applied repeatedly until no new points-to edges can be added to the points-to graph. The

[3] A flow-insensitive analysis ignores the flow of control between program points. A context-insensitive analysis does not distinguish between different invocations of a method. Flow-sensitive analyses are more precise and costly than flow-insensitive ones. Similarly, context-sensitive analyses are more precise and costly than context-insensitive ones.

resulting points-to graph is examined to determine the solution for the corresponding class analysis *AND*.

Example Consider the sample program in Figure 6. Due to the object creation statements at line 2 and line 3, points-to edges (p, o_1) and (q, o_2) are added to the graph. At line 4, p points to object o_1. Based on the class of o_1 and the method identifier $X.set$, the analysis determines that the method invoked by the virtual call at line 4 is $X.set$. Thus, implicit parameter $this$ of method $X.set$ is set to point to o_1, and formal parameter r is set to point to all objects in the points-to set of actual parameter q. In this case, the analysis infers two new points-to edges: $(this, o_1)$ and (r, o_2). Finally, when the rule for $this.f = r$ is applied at line 1, fields f of all objects in the points-to set of $this$ are set to point to all objects in the points-to set of r. In this example, as a result of this rule, the analysis infers points-to edge $(\langle o_1, f\rangle, o_2)$. All points-to edges are shown in the final points-to graph in Figure 6. The corresponding class analysis *AND* infers the following sets:

$$Cs(X.set.r) = \{Y\} \quad Cs(p) = \{X\} \quad Cs(q) = \{Y\}$$

Object sensitivity [16] is an approach for context sensitivity of flow-insensitive points-to analysis for object-oriented languages. The key idea is that every instance method and constructor is analyzed separately for each of its receiver objects. *OBJ-PT* is an object-sensitive version of *AND-PT*. Because *OBJ-PT* distinguishes contexts of invocations associated with distinct receiver objects, it is able to avoid merging the effects of instance methods and constructors over all possible receivers.

Example Recall the set of statements from Figure 6. Suppose that the following statements are added at lines 5, 6, 7, and 8 in method `main`:

```
5  s3 :  X p2 = new X();
6  s4 :  Z q2 = new Z();
7        p2.set(q2);
8        Y q3 = p2.f;
```

When these statements are analyzed using *AND-PT*, there are spurious points-to edges $(\langle o_1, f\rangle, o_4)$, $(\langle o_3, f\rangle, o_2)$, and (q_3, o_2). This imprecision affects class analysis. For example, *AND* erroneously infers that the set of possible classes for variable q_3 is $\{Y, Z\}$. *OBJ-PT* avoids merging the effects of method $X.set$ for receivers o_1 and o_3, and only creates points-to edges $(\langle o_1, f\rangle, o_2)$, $(\langle o_3, f\rangle, o_4)$, and (q_3, o_4). Thus, the corresponding class analysis *OBJ* infers that the possible set of classes for q_3 is $\{Z\}$.

4 Empirical Results

We performed experiments on nine publicly available Java programs, ranging in size from 177KB to about 1MB of bytecode. The set includes programs from the SPEC JVM98 suite, other benchmarks used in previous work on analysis for Java, as well as programs from an Internet archive (`www.jars.com`) of popular publicly available Java applications. All experiments were performed on a 360MHz Sun Ultra-60 machine with 512MB memory.

Program	User Class	Size (KB)	Whole-program		
			Class	Method	Stmt
mpegaudio	62	176.8	608	3531	71712
jjtree-1.0	72	272.0	620	4078	79587
sablecc-2.9	312	532.4	864	5151	82418
javac	182	614.7	730	4470	82947
creature	65	259.7	626	3881	83454
mindterm1.1.5	120	461.1	686	4420	90451
soot-1.beta.4	677	1070.4	1214	5669	92521
muffin-0.9.2	245	655.2	824	5253	94030
javacc-1.0	63	502.6	615	4198	102986

Table 1. Characteristics of the data programs.

Table 1 shows some characteristics of the data programs. The first two columns show the number of user (i.e., non-library) classes and their bytecode size. The next three columns show the size of the program, including library classes, after using *CHA* to filter out irrelevant classes and methods. We use the Soot framework (`www.sable.mcgill.ca`) to process Java bytecode and to build an intermediate representation of the program [26]. The last column in Table 1 shows the number of statements produced by Soot after processing the bytecode.

4.1 Precision

In our experiments we measured the impact of class analysis on ORD construction with respect to two metrics: (i) the number of edges in the ORD and (ii) the average size of the class firewall. We also measured the impact of class analysis on the number of edges in the ExtORD.

Java programs typically contain a large portion of standard library code. We believe that in most cases one can assume the correctness of the library code, and therefore interactions between library classes and interactions between library classes and user classes do not need to be tested. In order to assess the impact of the different class analyses on determining the dependencies between user classes, we extract the *user-class-related* portion of the ORD/ExtORD computed by the algorithm in Figure 5. The nodes of the user-class-related subgraph are all nodes representing user classes, and the edges are all edges connecting user class nodes. For the remainder of this paper the terms ORD and ExtORD will be used to refer to these corresponding user-class-related subgraphs.

Program	*CHA*	*AND*	*OBJ*	Reduction	
				AND	*OBJ*
mpegaudio	437	232	229	46.9%	47.6%
jjtree	604	351	351	41.9%	41.9%
sablecc	38643	18023	5531	53.4%	85.7%
javac	12292	7672	7654	37.6%	37.7%
creature	619	272	232	56.1%	62.5%
mindterm	1028	423	420	58.9%	59.1%
soot	76223	32581	32479	57.3%	57.4%
muffin	11431	2522	2522	77.9%	77.9%
javacc	780	554	554	29.0%	29.0%
			Avg	51.0%	55.4%

Table 2. ORD size.

Program	*CHA*	*AND*, *OBJ*	Reduction *AND*, *OBJ*
mpegaudio	38	22	42.1%
jjtree	53	41	20.8%
sablecc	283	270	4.6%
javac	152	129	15.1%
creature	44	42	4.5%
mindterm	82	68	17.1%
soot	382	362	5.2%
muffin	192	136	29.2%
javacc	32	20	37.5%
		Avg	19.6%

Table 3. Class firewall size.

4.1.1 ORD Size

The size of the ORD is an indication of the precision of the diagram; more precise diagrams contain fewer edges. The first three columns of Table 2 show the number of edges computed by *CHA*, *AND*, and *OBJ*. The improvements of *AND* and *OBJ* over *CHA* are shown in the last two columns of the table. On average, *AND* reduces the number of ORD edges by 51% and *OBJ* by 55%.

These results show that class analyses based on points-to analyses can significantly reduce the number of spurious dependence edges. The improved precision may lead to less time and effort spent on cycle breaking and stub construction when the ORD is used to define a test order in integration testing or a retest order in regression testing.

4.1.2 Class Firewall Size

The size of the class firewall indicates how suitable the ORD is for use in regression testing. Class firewalls computed from a more precise ORD contain fewer classes. The first two columns in Table 3 show the average class firewall sizes for our benchmarks. For each user class in a program, we calculated the size of its firewall. Then we took the average of these firewall sizes over all the user classes in the program. The last column in the table shows the improvements of *AND* and *OBJ* over *CHA*.[4] On average, the more precise class analyses reduce the average class firewall size by almost 20%.

These results show that class analysis based on points-to analysis produces substantially smaller class firewalls, which may result in less work and less time and effort spent on regression testing. Savings may occur because classes *not* affected by the change which triggered the regression testing will not be retested.

[4]The numbers for *AND* and *OBJ* are the same.

4.1.3 ExtORD Size

In order to estimate the impact of the different analyses on coverage analysis, we computed the size of the ExtORD. This size is an indication of the precision of the diagram; more precise diagrams contain fewer dependence edges (i.e., fewer spurious edges).

The first three columns in Table 4 show the number of edges computed by *CHA*, *AND*, and *OBJ* respectively. The percentage improvements for *AND* and *OBJ* over *CHA* are shown in the last two columns. On average, *AND* reduces the number of edges in the ExtORD by 36% and *OBJ* by 39%.

We draw two conclusions from these results. First, the results computed by *CHA* are very imprecise and this leads to a significant number of spurious dependence edges. Attempting to exercise statements in a way that triggers these impossible dependences in order to achieve high coverage will lead to substantial waste of time and effort. Second, this significant reduction shows that *AND* and *OBJ* are good candidates for use in tools for coverage analysis.

4.2 Points-to Analysis Cost

The measurements of points-to analysis cost are presented in Table 5. The first two columns show the running time and memory usage of *AND-PT*. The last two columns show the cost of *OBJ-PT*. These empirical results show clearly that the two points-to analyses are practical in terms of running time and memory consumption. Thus, these analyses are realistic candidates for use in software engineering tools.

5 Related Work

The firewall approach was proposed by White et al. for regression testing of procedural code [27, 28]. Kung et al. [13] define the Object Relation Diagram (ORD) and adapt the firewall approach for object-oriented languages.

Program	*CHA*	*AND*	*OBJ*	Reduction	
				AND	*OBJ*
mpegaudio	1411	1009	999	28.5%	29.2%
jjtree	7796	7164	7163	8.1%	8.1%
sablecc	87632	38023	11626	56.6%	86.7%
javac	66072	49208	49077	25.5%	25.8%
creature	3932	2962	2918	24.7%	25.8%
mindterm	4751	2525	2474	46.9%	47.9%
soot	153418	75026	74722	51.1%	51.3%
muffin	21186	6203	6203	70.7%	70.7%
javacc	8593	7907	7907	8.0%	8.0%
			Avg	35.6%	39.3%

Table 4. ExtORD size.

Program	*AND-PT*		*OBJ-PT*	
	Time [sec]	Memory [MB]	Time [sec]	Memory [MB]
mpegaudio	32.0	53	29.7	52
jjtree	23.7	53	24.4	52
sablecc	136.6	112	73.1	94
javac	973.4	122	956.9	122
creature	176.1	90	126.3	87
mindterm	82.3	91	93.0	88
soot	146.1	130	171.8	131
muffin	236.3	144	214.0	133
javacc	165.2	110	169.5	112

Table 5. Running time and memory usage.

However, Kung et al.'s work does not consider the effects of dynamic binding of polymorphic variables.

Labiche et al. [14] propose an approach for correcting the problem with dynamic bindings. Their work uses the structure of the class hierarchy to determine possible bindings. Our approach uses more precise class analysis and constructs ORDs with significantly fewer spurious dependence edges, compared to ORDs constructed by examining the class hierarchy.

Work by Tai and Daniels [24] and Jeron et al. [12] concentrates on approaches for cycle breaking in the ORD. This work addresses the following question: given the diagram, what cycles should be removed so that the minimum number of stubs will need to be constructed. Our work concentrates on improving the precision of the ORD, which potentially leads to fewer dependence cycles.

There are many existing class analyses [17, 1, 2, 18, 9, 4, 11, 8, 6, 19, 21, 22, 23, 25, 15, 20, 10, 16] that have been used in the context of optimizing compilers and software engineering tools for object-oriented programming languages. Our work is the first one to investigate the use of these analyses for the purposes of ORD/ExtORD construction.

6 Conclusions and Future Work

We have shown how class analysis can be used to construct the Object Relation Diagram (ORD) of a program. Our empirical results demonstrate that using precise class analyses can result in substantially more precise ORDs, compared to ORD construction based on the structure of the class hierarchy.

We have defined the Extended Object Relation Diagram (ExtORD), which is a version of the ORD suitable for use in coverage analysis. Any method developed for ORD construction can be easily extended to construct the ExtORD. Our experiments show that precise class analyses can substantially improve the precision of the ExtORD.

In our future work we plan to use class analysis for ORD construction in the context of tools for integration and regression testing. Our goal is to develop algorithms for regression test case selection based on precise ORDs. We also plan to build integration coverage analysis tools based on the ExtORD. Finally, we would like to investigate the use of these dependence diagrams for the purposes of program understanding and reengineering.

7 Acknowledgments

We would like to thank the ICSM reviewers for their helpful comments. This research was supported by NSF grant CCR-9900988.

References

[1] O. Agesen. Constraint-based type inference and parametric polymorphism. In *Static Analysis Symposium*, LNCS 864, pages 78–100, 1994.

[2] O. Agesen. The cartesian product algorithm. In *European Conference on Object-Oriented Programming*, LNCS 952, pages 2–26, 1995.

[3] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[4] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, 1996.

[5] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[6] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[7] J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[8] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[9] A. Diwan, J. B. Moss, and K. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 292–305, 1996.

[10] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, Nov. 2001.

[11] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 108–124, 1997.

[12] T. Jeron, J.-M. Jezequel, Y. L. Traon, and P. Morel. Efficient strategies for integration and regression testing of OO systems. In *International Symposium on Software Reliability Engineering*, pages 260–269, 1999.

[13] D. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Class firewall, test order and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, 8(2):51–65, 1995.

[14] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck, and M.-H. Durand. Testing levels for object-oriented software. In *International Conference on Software Engineering*, pages 136–145, 2000.

[15] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, 2001.

[16] A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analysis for Java. In *International Symposium on Software Testing and Analysis*, 2002.

[17] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 146–161, 1991.

[18] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–340, 1994.

[19] C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, Dec. 1999.

[20] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java based on annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 43–55, 2001.

[21] E. Ruf. Effective synchronization removal for Java. In *Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[22] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, U. Passau, Sept. 2000.

[23] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 264–280, 2000.

[24] K.-C. Tai and J. F. Daniels. Interclass test order for object-oriented software. *Journal of Object-Oriented Programming*, 12(4):18–25, 1999.

[25] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 281–293, 2000.

[26] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, pages 18–34, 2000.

[27] L. White and H. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *International Conference on Software Maintanence*, pages 262–271, 1992.

[28] L. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha. Test manager: A regression testing tool. In *International Conference on Software Maintanence*, pages 338–347, 1993.