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 important applications in integration testing, integration coverage analysis and regression testing. The precision of the ORD, that is how closely it reflects what actually can occur during program execution, directly affects the efficiency (and therefore the practicality) of its usage. This paper makes three key contributions to the use of ORDs in testing. 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 resulting in 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 and most importantly, we show empirically that relatively precise class analyses can be used to construct precise ORDs and ExtORDs, whose size improvement over earlier techniques is on average 56%-60% or 34%-38% respectively, (depending on the class analysis). In addition, more precise class analyses substantially reduce the size of *class firewalls*, that set of classes affected by a change to a particular class, which in turn diminishes the effort required for regression testing.

1 Introduction

Object-oriented systems are characterized by complex interclass interactions. The goal of *integration testing* is to reveal faults in classes that cause some class 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 *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 goal of *integration coverage analysis* is to show that sufficiently many interclass interactions are covered by the test set. 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).

Kung et al. [12] define the *Object Relation Diagram (ORD)* to model the dependences between program classes. The ORD of program P is a directed graph, where the nodes represent the classes in P, and the edges represent the dependences between classes. There are three kinds of edges labeled I for *Inheritance*, Ag for *Aggregation* and As for *Association.* (Several diagrams are shown in Figure 2 in Section 2). The *class firewall* of a given class C is the set of classes reachable backwards on the ORD from the node representing C; intuitively, it represents the set of classes that might be affected if a change is made to C. The ORD and the class firewall can be used to aid in testing of object-oriented programs. The ORD can be used to define an efficient test order for integration testing. It can be used to find the set of classes affected by a program change and to define a regression test order for this set of classes.

One disadvantage of existing ORDs is that there is at most edge of each kind between two classes. Therefore, coverage analyzers cannot use the ORD to find out which specific statement triggered 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 and thus, test coverage of such statements can be distinguished. In general, any method used to construct the ORD from program code can be easily extended to construct the ExtORD.

Imprecise ORDs and ExtORDs contain spurious dependence edges, representing interclass dependences that are impossible and do not correspond to any program execution. Imprecision may lead to a lot of (human) time wasted on integration or regression testing. When the class firewalls are imprecise, (i.e., too large), time is wasted on retesting unaffected classes. Imprecision leads to a tester spending time on trying to execute a statement which triggers a dependence that cannot happen. Much of this time and effort would have been saved if the ORD or the ExtORD were more precise (i.e., contained fewer spurious edges).

Because of the wide range of applications of these dependence diagrams, it is important to investigate approaches for efficient construction of precise ORDs and ExtORDs. *Class analysis* is one popular form of static program analysis of object-oriented languages and 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. This information is a *necessary prerequisite* for the construction of the ORD and the ExtORD and the precision of the ORD and the ExtORD directly depends on the solution of the class analysis problem. There are many class analyses with different tradeoffs between cost and precision, developed primarily in the context of compilation of object-oriented programming languages.

In this paper we define a generalized algorithm for ORD construction, which is parameterized by a class analysis. The algorithm allows the developer to build ORDs of differing degrees of precision. We show that certain relatively precise class analyses can be used to construct the ORD or ExtORD efficiently and result in *substantially more precise* ORDs or ExtORDs. 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.

We have evaluated three class analyses on a set of 9 realistic Java programs, ranging in size from 66Kb to 1 Mb of bytecode. We have estimated the impact of the three class analyses on the precision of the ORD. We compared the improvements of the two more precise analyses over the least precise analysis. To the best of our knowledge, before our work the least precise analysis (i.e., class hierarchy analysis) was the only method used by ORD construction to account for dependences due to dynamic binding. With respect to the *ORD size* (defined as the number of edges), the two more precise analyses compute substantially smaller ORDs, removing significant numbers of spurious dependence edges. This reduction shows that using these analyses in tools that assist developers in integration or regression testing may result in substantial savings in human time and effort.

Our results also show that the precise analyses reduce the average size of the class firewall significantly. Thus, substantially less time and effort is expected to be needed for regression testing if the precise ORD is used to compute the firewall.

In addition, we measured the size of the ExtORD, which shows how useful the diagram may be for coverage analysis. The more precise analyses produce substantially smaller ExtORDs. Thus, if one of the more precise ExtORDs is used, substantially less time will be spent trying to exercise dependences that never happen, than if the least precise ExtORD is used.

Even though most of the programs are large, the more precise (and costly) analyses run in less than four minutes on all but one program. For all programs these analyses use less than 150Mb of memory. Thus, the more precise analyses can run in practical time and space on large programs. This practicality makes them realistic candidates for use in software tools.

Contributions The contributions of our work are the following:

- We define the Extended Object Relation Diagram (ExtORD) for use by integration coverage analyzers. In general, 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 developers to control the cost versus precision tradeoffs of ORD construction by varying the class analysis.
- Most importantly, we show that the two more precise analyses improve substantially over the least precise analysis in terms of the size of the ORD, the size of the ExtORD and the size of the class firewall, while remaining efficient and practical. These results show that using these analyses in software tools for ORD construction may result in substantial savings in human time and effort for client applications.

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 the precision of the ORD. Section 3 discusses three specific class analyses and presents a general algorithm for ORD construction. 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

The *Object Relation Diagram (ORD)* models the dependences between program classes. The ORD of program P is a directed graph, where the nodes represent the classes in P, and the edges represent the dependences between classes. An edge from node A to node B represents the fact that class A depends on class B. Intuitively, there are dependences due to inheritance, constructor calls (i.e., when a constructor of class B is called in method/constructor defined in class A), instance method calls, field accesses, etc.

This section is organized as follows. Section 2.1 describes applications of ORDs. Sections 2.2 and 2.3 summarize existing work on ORD construction. Section 2.4 argues the importance of ORD precision.

2.1 Applications of Object Relation Diagrams

The ORD can be used in integration testing and regression testing. Once constructed, the ORD for a set of classes can be reused by various clients at no additional development 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 in classes 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 derive 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, 12, 13] aim at minimizing the number of stubs, because stub construction requires significant human effort. This test order can be derived from the ORD 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 dependent classes and so on.

Top-down integration testing strategies [5] start testing at the classes at the top-level of the control hierarchy and use stubs for the necessary server classes. Usually a large number of stubs is needed to set up a test case. The ORD can be used to determine the necessary stubs at each level. For example, a dependence edge from class A to class B, indicates that running a test case on class A may require that the behavior of class B in the context of class A is simulated by stub(s).

Coverage Analysis The goal of coverage analysis is to ensure the quality of the test set, (i.e., the test set covers a certain portion of the program code). Integration testing focuses on faults due to complex interclass dependences. Thus, it is important to make sure that the set of test cases covers the code which generates interclass dependences. Because of polymorphism, a single statement may trigger different interclass dependences. Therefore, the test set must cover such statements in a way that exercises *sufficiently* many of these dependences. An important exit criterion for many integration testing strategies is that all interclass *call pairs* are exercised [5] (the term call pair, defined in [5] refers to a client and a server that have a call between them). A version of the ORD can be used by coverage analyzers to determine the set of interclass method calls (call pairs) that need to be exercised by the set of test cases in order to satisfy this requirement. Others [19] propose (without specifying how) that every dependence edge between two classes should be covered by a test case.

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 [10, 9]. 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 the affected classes needs to be retested in order to ensure that their behavior satisfies the requirements for the class. Two fundamental problems in regression testing are: (i) how to identify the affected classes and (ii) how to perform retesting of the affected classes *efficiently*. The ORD can be used in regression testing to identify the set of affected classes. All classes in the class firewalls of 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 and leads to efficient retesting strategy, because stub construction requires significant human effort. Analogously 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 must be tested before the classes that depend on it; when the classes that depend on A are tested, stubs are not required for A because it has already been retested).

2.2 Kung et al.'s ORD

Kung et al. [12] define the Object Relation Diagram using three kinds of dependence edges:

Inheritance There is an edge labeled I from class B to class A if and only if B is a direct subclass of A. For the rest of the paper we use the notation $(\langle B, I\rangle, A)$ to denote an edge labeled I from B to A.

Aggregation There is an edge $(\langle B, Ag\rangle, A)$ if and only if constructor invocation `new A(...)` appears in a statement contained by method defined in 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 method or constructor in class B. We refer to such an association as a *parameter* association.
- Indirect field access `r.f` appears in a statement contained by method or constructor defined in class B and the declared class type of reference variable r is A ($r \neq$ `this`). We refer to such an association as a *field access* association.

[1]According to [12] aggregation can be (i) static, due to encapsulated non-pointer fields and (ii) dynamic, due to constructor invocation. Since all fields in Java are references, (i.e., pointers to objects), and thus static aggregation is not possible, we simplify the definition by including only dynamic aggregation.

```
   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: Example set of statements.

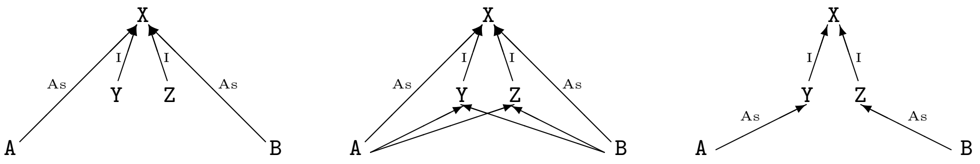
- Method invocation `r.m(...)` appears in a statement contained by method or constructor in class B and the declared class type of r is A ($r \neq$ `this`). We refer to such an association as a *method call* association.

Figure 2 (a) shows the ORD for the simple program on 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 in Figure 2(a).

The *class firewall*, denoted by $CFW(C, ORD)$ for class C is defined as the set of classes reachable from the node corresponding to class C in the transpose of ORD[2] [12, 13]. 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 on Figure 2(a), $CFW(X) = \{X, Y, Z, A, B\}$ and $CFW(Y) = \{Y\}$.

We define the *Extended Object Relation Diagram (ExtORD)* which will make the relation diagram more informative and more useful for code coverage analysis tools. The ExtORD contains *annotated* association and aggregation edges. There is an annotated association $As : s_i$, if and only if s_i is a program statement which triggers a field access or a method call association. Similarly, there is an annotated aggregation $Ag : s_i$, if and only if s_i is an object creation statement. There is a non-annotated association As which denotes parameter association. Thus, the ExtORD may contain multiple association edges between two nodes, because the ExtORD makes explicit which program

[2]The transpose of graph $G = (V, E)$ is a graph $G^T = (V, E^T)$, where $E^T = \{(\langle u, l\rangle, v) | (\langle v, l\rangle, u) \in E\}$.



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

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

statement triggers the association. 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 association between class A and class X. It is easy to see how coverage analyzers can make use of the Extended Object Relation Diagram. The ExtORD reflects *all* interclass dependences and the statements which *trigger* interclass dependences. Thus, the ExtORD identifies the statements that need to be exercised by a high quality test set.

2.3 Labiche et al.'s ORD

In order to correct the problem arising because of possible dynamic bindings of polymorphic variables, Labiche et al. [13] propose to augment the ORD computed by Kung et al.'s approach with additional edges representing dynamic relations. If there is an edge labeled As from A to B, there are additional edges with the same label from all subclasses of A (including transitive subclasses) to B. 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. This approach is equivalent to modifying Kung et al.'s rules for association inference to take into account the set of possible dynamic bindings of polymorphic variables that trigger the association(e.g. `r` in virtual call `r.m()`). In general, this set can be determined by analysis of the program code. Labiche et al. use analysis of the class hierarchy rooted at the declared class type for such variable.

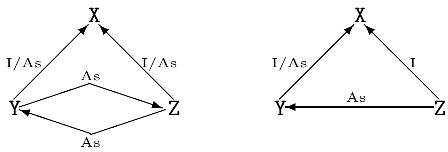
Using this analysis, the ExtORD can be augmented in similar fashion. If there is an edge labeled $As : s_i$ from A to B, there are edges labeled $As : s_i$ from all subclasses of A (including transitive subclasses) to B. For the example set of statement in Figure 1, there are additional edges $(\langle A, As{:}4\rangle, Y)$ and $(\langle A, As{:}4\rangle, Z)$. In order to achieve good coverage of the code which generates 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) may need to be exercised several times in order to achieve coverage for all dynamic interclass dependences that may result from such statement.

2.4 The Disadvantages of Imprecise Analysis

Clearly, the ORD computed by Kung et al. omits dependences and may result in incomplete testing. The ORD computed by Labiche et al. is *imprecise*. This imprecision results from imprecise analysis of the possible dynamic bindings for polymorphic variables. In Figure 1 class A does not depend on class Z because at run time `A.A.xa` on line 2 and `A.m.xa` on 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).

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

Figure 3: Set of statements.



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

Figure 4: ORDs corresponding to Figure 3

Analysis imprecision which results in spurious dependences can impair the usefulness of testing tools which make use of the ORD 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 [11, 19] and (ii) breaking cycles inevitably requires that one or more stubs are constructed. Consider the ORDs in Figure 4 which correspond to the set of statements in Figure 3. The imprecise diagram in Figure 4(a) contains a dependence cycle between Y and Z.[3] This cycle is caused by the imprecise class analysis which interprets a dependence on class X additionally into a dependence on each of its subclasses Y, Z. Therefore, the test order cannot be determined without cycle breaking. In contrast, the precise diagram in Figure 4(b) clearly shows that the right test order is X, Y, Z.

If the ORD is used in top-down integration testing, an imprecise diagram may imply the existence of dependences that are impossible. For example, the imprecise ORD in Figure 4(a) implies that setting up test cases for class Y requires stubs simulating the behavior of Z in the context of Y, while in fact Y does not depend on Z.

In coverage analysis, imprecision can lead to inadequate

[3]There are loop association edges at Y and Z. We omit these edges from the picture for clarity. Clearly, loop edges can be ignored when the ORD is used for test order definition.

coverage: an instance method call from class A on a receiver of class B may not be exercised because there is no dependence between these two classes, and not because the test set is incomplete. For example, in the ExtORD computed by the analysis of Labiche et al. for the set of statements in Figure 1, the following three association edges correspond to the method call at line 4:

$$(\langle A, As{:}4\rangle, X) \qquad (\langle A, As{:}4\rangle, Y) \qquad (\langle A, As{:}4\rangle, Z)$$

The programmer will attempt to exercise the call with receivers from class X and Z and will spend valuable time showing that variable `A.m.xa` cannot refer to objects of any other class but Y.

In regression testing, an imprecise ORD can lead to large class firewalls, (i.e., substantially more than necessary code may be selected for retesting).

In all cases, imprecision of the ORD results in human effort spent for nothing. Breaking spurious dependence cycles, executing a statement in a manner that triggers nonexistent interclass dependences, retesting parts of the code that are not affected by the change, all of these waste the tester's time. Therefore, it is important to investigate approaches for constructing more precise ORDs.

3 Class Analysis for ORD construction

Recall from Section 2 that unlike inheritance and aggregation dependences, determining association dependences requires analysis of the possible dynamic bindings of certain polymorphic variables. To determine all classes for which statement `p.f=q` triggers field associations with the enclosing class of the statement, one needs to find out the possible dynamic bindings of reference variable `p`, (i.e., the classes of all objects `p` may refer to at run-time). Similarly, for method call associations one needs the set of all possible classes for the receiver object, and for parameter associations one needs the set of all classes of objects that can be bound to explicit parameters.

Exactly this information is computed by class analysis, which has been studied extensively in the context of compiler optimization. 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 [16, 1, 6, 4, 8, 20, 18]. More precise class analyses result in fewer spurious dependences and therefore more precise ORDs and ExtORDs. For the example set of statements in Figure 3, the set of possible classes of all objects that may be bound to variable `xy` at line 1, equals $\{X\}$. Set $\{X, Y, Z\}$ is a valid approximation because it includes the only possible run-time class X, but it is less precise because it includes Y and Z and objects of classes Y or Z cannot be bound to `xy`.

In Section 3.1 we discuss class hierarchy analysis (CHA) which is a fundamental class analysis [6]. Section 3.2 and Section 3.3 discuss two more precise class analyses derived from static analyses we have developed in our previous work. Section 3.4 presents a general algorithm for ORD construction, parameterized by a class analysis. Thus, the algorithm allows the developer to build ORDs of differing degrees of precision.

3.1 Class Hierarchy Analysis

CHA is an inexpensive analysis that determines the required set of classes for program variables by examining the class hierarchy of the program. For given variable r of declared class type C, the set of possible classes of objects that may be bound to r at run-time is the set containing C and all direct and transitive subclasses of C.[4] For example, CHA computes the following sets of classes, denoted by $Cs(x)$ for the variables in Figure 5.

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

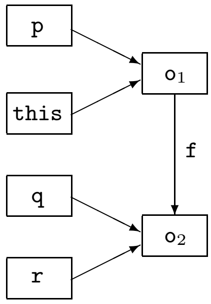


Figure 5: Sample program and its points-to graph.

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

Clearly, the method used by Labiche et al. to determine the possible dynamic bindings of polymorphic variables is essentially CHA.

3.2 Class Analysis Based on Context-Insensitive Points-to Analysis

We examine two class analyses derived from *points-to analyses* developed in our previous work [17, 15]. 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 5. The points-to graphs contain two kinds of edges. The two different kinds of edges are shown in Figure 5: (i) edge (p, o_1) shows that reference variable p points to object o_1, where object name o_1 represents all objects that may be created at allocation site s_1 during execution of the program and (ii) *field* edge $(\langle o_1, f\rangle, o_2)$ shows that field f of object o_1 points to object o_2.

Using the points-to graph computed by the points-to analysis, the set of possible classes of objects that may be bound to r can be derived by examining the class types of the objects in the points-to set of r.

This section presents a points-to analysis which is flow-insensitive and context-insensitive and Section 3.3 presents a context-sensitive version of the analysis in this section.[5]

The context-insensitive points-to analysis is derived from Andersen's analysis for C [3]. A detailed description of the analysis and its implementation can be found in [17]. The analysis is defined in terms of three sets. Set R contains all reference variables in the analyzed program (including static variables). Set O contains names for all objects created at

[4] Excluding abstract classes

[5] 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. Intuitively, 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.

object allocation sites; for each allocation site s_i, there is a separate object name $o_i \in O$. Set F contains all instance fields in program classes.

For brevity, we only discuss the kinds of statements listed below. Other kinds of statements (e.g. calls to constructors and static methods) are handled in a similar fashion.

- Direct assignment: `l = r`
- Instance field write: `l.f = r`
- Instance field read: `l = r.f`
- Object creation: `l = new C`
- Virtual invocation: $\texttt{l = r}_0\texttt{.m(r}_1\texttt{,...,r}_k\texttt{)}$

At a virtual call, name `m` uniquely identifies a method in the program. This method is the *compile-time* target of the call, and is determined based on the declared type of $\texttt{r}_0$ [7, Section 15.11.3]. At run-time, the invoked method is determined by examining the class of the receiver object and all of its superclasses, and finding the first method that matches the signature and the return type of `m` [7, Section 15.11.4].

The analysis uses *rules* that add new edges to points-to graphs. Each rule represents the meaning of a program statement. Statement $l = r$ creates new points-to edges from l to all objects pointed to by r. 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. Analogously, statement $l = r.f$ creates new points-to edges from l to all objects to which field f of an object in the points-to set of r points. The rule for object creation $l = new\ C$ creates a points-to edge from l to a brand new object of class C.

For virtual call sites $l = r_0.m(r_1, ..., r_k)$, resolution is performed for every receiver object pointed to by r_0. A function *dispatch* uses the class of the receiver object and the compile-time target of the call to determine the actual method m_j invoked at run-time. Let variables $p_0, \ldots, p_n$ denote the formal parameters of the method m_j; variable p_0 corresponds to the implicit parameter `this`. Let variable ret_j denote the return values of m_j (we assume that each method has a unique variable that is assigned all values returned by the method; this can be achieved by inserting auxiliary assignments). A points-to edge is created from implicit parameter `this` of m_j to the receiver object. Subsequently, for each explicit parameter p_i, new points-to edges are created from p_i to all objects in the points-to set of the corresponding actual parameter r_i. The flow of values to the left-hand side of the call is reflected by inserting new points-to edges from l to all objects in the points-to set of ret_j. The rules are applied repeatedly until no new points-to edges are generated. This procedure results in a final points-to graph, which represents the solution computed by the analysis.

Example Consider the example program in Figure 5. 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, which is X and the compile-time target of the method, which is $X.set$, function *dispatch* determines that method $X.set$ is applied at line 4. 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 indirect field write 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 5.

Thus class analysis based on this points-to analysis infers the following sets for the example set of statements in Figure 5, an improvement over class hierarchy analysis.

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

3.3 Class Analysis Based on Object-Sensitive Points-to Analysis

Object sensitivity is an approach to context sensitivity for the points-to analysis described in Section 3.2. The key idea in this approach is to analyze a method separately for each of the objects on which this method is invoked. Augmenting Andersen's analysis in Section 3.2 with object sensitivity results in a more precise points-to analysis, which may lead to more precise class analysis and more precise ORDs. For brevity we omit the formal discussion of object sensitivity and include only an intuitive description of the analysis. Details on object sensitivity can be found in [15].

Intuitively, object-sensitive analysis distinguishes implicit parameter `this` for distinct receiver objects. Thus it avoids merging the effects of instance methods and constructors over all possible receivers. To illustrate the intuition behind object-sensitivity recall the set of statements from Figure 5. 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 context-insensitively, using the rules described in Section 3.2, 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, class analysis based on context-insensitive analysis erroneously infers that the set of possible classes for variable q_3 on line 8 is $\{Y, Z\}$. Object-sensitive analysis avoids merging the effects of method `X.set` for receivers o_1 and o_3. The corresponding class analysis infers precisely edges $(\langle o_1, f\rangle, o_2)$, $(\langle o_3, f\rangle, o_4)$ and (q_3, o_4). Thus, class analysis based on object-sensitive points-to analysis infers that the possible set of classes for q_3 is precisely $\{Z\}$.

3.4 ORD Construction

Figure 6 presents an algorithm, parameterized by a class analysis, which computes the ORD starting from empty ORD. $Cs(x)$ is the output of given class analysis. It denotes the set of classes which approximates the classes of all objects that may be bound to variable x. *EnCl* denotes the enclosing class of a given statement or method. Figure 6 shows the process of constructing the ORD by examining each statement. For example, for each indirect read statement $q = p.f$ (line 4 through 6), the algorithm adds association edges to the ORD from the node representing the class enclosing the statement to each node representing a class in the set of classes computed by the class analysis for variable p (if p is equal to `this`, no association edges are inserted).

For brevity we omit discussion of calls to static methods and access of static fields. Our implementation handles these cases properly.

input *Stmt*: set of statements
Methods: set of methods
Cs: $R \rightarrow \mathcal{P}(C)$
output *ORD*

foreach indirect write $s: p.f = q \in \mathit{Stmt}$ **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 indirect read $s: q = p.f \in \mathit{Stmt}$ **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 $s: l = p.m(...) \in \mathit{Stmt}$ **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 $m: T\, m(...)\, \{...\} \in \mathit{Methods}$ **do**
  foreach explicit formal parameter p **do**
    $\mathit{ORD} := \mathit{ORD} \cup \{(\langle \mathit{EnCl}(m), \mathit{As}\rangle, C) \mid C \in \mathit{Cs}(p)\}$
foreach object creation $s: p = \mathit{new}\ C \in \mathit{Stmt}$ **do**
  $\mathit{ORD} := \mathit{ORD} \cup \{(\langle \mathit{EnCl}(s), \mathit{Ag}\rangle, C)$

Figure 6: ORD construction, parameterized by class analysis. $\mathcal{P}(X)$ denotes the power set of X.

Program	User Class	Size (Kb)	Whole-program Class	Method	Stmt
javacup-0.10	33	127.3	581	3564	66463
jflex-1.2.2	54	198.2	608	3692	71198
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
mindterm1.1.5	120	461.1	686	4420	90451
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. First two columns show the number and bytecode size of user classes. Last three columns include library classes.

4 Empirical Results

The two points-to analyses are implemented using annotated inclusion constraints [17, 15]. We use the Soot framework (`www.sable.mcgill.ca`) to process Java bytecode [21]. The points-to analysis implementations are based on the Bane toolkit (`bane.cs.berkeley.edu`) for constraint-based program analysis [2]. We denote by *ObjSens* the class analysis based on the object-sensitive points-to analysis described in Section 3.3; the class analysis based on the context-insensitive analysis described in Section 3.2 is denoted by *And*.

All experiments were performed on a 360MHz Sun Ultra-60 machine with 512Mb physical memory. The reported times are the median values out of three runs. We used 9 publicly available data programs, ranging in size from 66Kb 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.

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. The last column shows the number of statements produced by Soot after processing the bytecode.

Program	CHA	*And*	*ObjSens*	Reduction *And*	Reduction *ObjSens*
javacup	569	160	160	71.9%	71.9%
jflex	620	191	191	69.2%	69.2%
mpegaudio	387	184	181	52.5%	53.2%
jjtree	563	311	311	44.8%	44.8%
sablecc	38392	17782	5290	53.7%	86.2%
javac	12194	7596	7578	37.7%	37.9%
mindterm	1027	429	426	58.2%	59.5%
muffin	11297	2408	2408	78.7%	78.7%
javacc	739	519	519	39.8%	39.8%
			Avg	56.3%	60.1%

Table 2: ORD size. The first three columns show the number of association and aggregation edges for the corresponding class analysis. The last two columns show the percentage reduction in the number of edges over CHA (derived from the first three columns).

4.1 ORD Precision

Java programs contain a large portion of standard library code. Relations between library classes as well as relations between user classes and library classes are irrelevant to integration testing and regression testing. We believe that application programmers and testers may assume the correctness of library code, that is, (i) stubs for library classes are not necessary and (ii) retesting of library classes is not necessary even though library classes may appear in the class firewall of an altered user class.

In order to assess the impact of the different class analyses on computing the relations between user classes, we extract the *user-class-related* portion of the ORD/ExtORD computed by the algorithm in Figure 6. The nodes of the user-class-related subgraph are the nodes representing user classes, and the edges are edges connecting two nodes representing user classes. For the remainder of this paper the terms ORD and ExtORD will be used to refer to these corresponding user-class-related subgraphs.

We measure the impact of class analysis on ORD construction with respect to two metrics: (i) ORD size, defined as the number of association and aggregation edges and (ii) the average size of the class firewall. We measure the impact of class analysis on ExtORD construction with respect to the number of association and aggregation edges.

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. In order to evaluate the impact of the different class analyses on the size of the ORD, we considered the number of aggregation and association edges. Inheritance edges are determined by the class hierarchy; thus, class analysis is irrelevant. The number of edges computed by CHA, *And* and *ObjSens* is shown in the first three columns of Table 2. The improvements of *And* and *ObjSens* over CHA are shown in the last two columns of Table 2. On average, *And* reduces the number of edges by 56% and *ObjSens* by 60%.

One surprising result is that for the majority of programs *And* and *ObjSens* are practically the same. We believe that this is due to the fact that we use a restricted form of object sensitivity, and not the full-blown version described in [15].

Program	CHA	*And*	Reduction
			And
javacup	27	16	40.7%
jflex	35	30	14.3%
mpegaudio	38	23	39.5%
jjtree	53	42	20.8%
sablecc	283	272	3.9%
javac	157	139	11.5%
mindterm	82	68	17.1%
muffin	192	141	26.5%
javacc	32	21	34.4%
		Avg	23.2%

Table 3: Class Firewall. The first two columns show the average class firewall size for the corresponding class analysis. The last two columns show the percentage reduction over CHA.

One exception is `sablecc`. The programmers of `sablecc` implement a pair of set/get methods in a class close to the root of a deep and wide inheritance hierarchy. *And* merges the effects of these methods for each object on which they are invoked. *ObjSens* avoids merging these effects and this leads to substantial precision improvements.

These results show that class analyses based on points-to analysis reduces the number of spurious dependence edges significantly. This precision improvement shows the potential for reductions in (i) the number of dependence cycles in the ORD and thus the number of unnecessary stubs resulting from cycle breaking and (ii) the number of unnecessary stubs that may need to be constructed if the program is tested in top-down manner. The improved precision may lead to less human time and effort spent on stub construction.

4.1.2 Class Firewall

The class firewall of class C ($CFW(C, ORD)$) is defined as the set of classes reachable from C in the transpose of the ORD. The average size of the class firewall is an indicator of the usability of the ORD in regression testing. A class firewall which is computed based on a more precise ORD contains 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 firewalls over all the user classes in a program. The result is reported as the firewall size (on average) in Table 3. The last column shows the improvements of *And* and *ObjSens* over CHA.[6] On average the more precise class analyses reduce the average class firewall size by more than 23%.

Given the results in Section 4.1.1, the fact that *And* and *ObjSens* produce the same class firewalls is not surprising. The smallest improvements are shown for `sablecc` and `javac`. These programs have deep and wide inheritance hierarchies. Their objects interact with large number of objects of different classes. It is not clear how often such applications occur in practice.

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

[6]The numbers for *And* and *ObjSens* are the same.

Program	CHA	*And*	*ObjSens*	Reduction	
				And	*ObjSens*
javacup	1693	1208	1197	28.6%	29.3%
jflex	2141	1306	1306	39.0%	39.0%
mpegaudio	1331	964	954	27.5%	28.3%
jjtree	7572	7061	7060	6.8%	6.8%
sablecc	63413	28517	9016	55.0%	85.8%
javac	59830	44834	44958	24.9%	25.1%
mindterm	4704	2527	2477	46.3%	47.3%
muffin	19614	5944	5944	69.7%	69.7%
javacc	8480	7893	7893	6.9%	6.9%
			Avg	33.9%	37.6%

Table 4: ExtORD size. The first three columns show the number of association and aggregation edges for the corresponding class analysis. The last two columns show the percentage reduction in the number of edges over CHA (derived from the first three columns).

4.2 ExtORD Size

In order to estimate the impact of the different analyses for ExtORD construction for coverage analysis, we computed the size of the ExtORD. Clearly, the size of the ExtORD is an indication of the precision of the ExtORD and its usefulness for coverage analysis tools. More precise diagrams contains fewer dependence edges, (i.e., fewer spurious edges). Analogously to the ORD, we computed the number of aggregation and association edges.

The first three columns in Table 4 show the number of edges computed by CHA, *And* and *ObjSens* respectively. The percentage improvements for *And* and *ObjSens* are shown in the last two columns. On average, *And* reduces the number of edges in the ExtORD by almost 34% and *ObjSens* by almost 38%.

We draw several conclusions from these results. First, the results computed by CHA are substantially imprecise. CHA leads to a significant amount of spurious dependence edges. Attempting to exercise statements in a way that triggers these dependences in order to achieve high coverage will lead to loss of substantial amount of human effort and time. Second, this substantial reduction shows that *And* and *ObjSens* can be usefully applied in tools for coverage analysis.

4.3 The Cost of Points-to Analysis

The measurements of analysis cost are presented in Table 5. The first two columns show the running time and memory usage of *And*. The last two columns show the cost of *ObjSens*[7]. The empirical results demonstrate that the two analysis are practical in terms of running time and memory consumption. For the majority of programs *ObjSens* has comparable performance to *And*. In certain cases (e.g., `sablecc`) the cost of the object-sensitive analysis is significantly lower than the cost of the context-insensitive analysis because the improved precision produces smaller points-to sets, which results in less work for the analysis. The results show clearly that the two points-to analyses are practical. Thus, these points-to analyses are realistic candidates for use in software tools.

[7]These times and memory usages are for the points-to analysis that does almost all the work of *And* and *ObjSens*. Given the points-to solution, we can associate the type of the objects pointed to, thus yielding the class analysis *And* (see Section 3.2).

Program	*And*		*ObjSens*	
	Time [sec]	Memory [Mb]	Time [sec]	Memory [Mb]
javacup	29.6	56	34.0	55
jflex	40.2	68	39.5	70
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
mindterm	82.3	91	93.0	88
muffin	236.3	144	214.0	133
javacc	165.2	110	169.5	112

Table 5: Running time and memory usage of the analyses.

5 Related Work

The class firewall approach was proposed by White et al. for regression testing of procedural code [22]. Kung et al. [12] define the Object Relation Diagram (ORD) and adapt the firewall approach for object-oriented languages. However, Kung et al.'s work does not consider the effects of dynamic binding of polymorphic variables on the ORD.

Labiche et al. [13] propose an approach for correcting the problem with dynamic binding from Kung's work. This work uses class hierarchy analysis for this purpose. Our approach uses more precise class analysis and constructs ORDs with significantly fewer spurious dependence edges.

Work by Tai and Daniels [19] and Jeron et al. [11] concentrate on approaches for cycle breaking in the ORD. This work addresses the 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 cycles.

There are many class analyses [16, 1, 6, 4, 8, 20, 18] which have been used for the compilation of object-oriented programming languages. Recently, a comparison of several class analyses for Java was presented by Liang and Harrold [14].

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 show that using precise class analysis derived from points-to analysis can reduce the number of spurious dependences in the ORD substantially over existing methods of ORD construction. Precise class analysis produces significantly smaller class firewalls as well.

We have developed 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. Precise class analysis reduces substantially the number of spurious dependence edges in the ExtORD over existing methods.

In our future work we plan to use class analysis for ORD construction in the context of tools for integration testing and regression testing. Our goal is to develop algorithms for regression test case selection based on precise ORDs. We plan to develop integration coverage analysis tools based on the ExtORD.

References

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

[2] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *International Workshop on Types in Compilation*, 1998.

[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] 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.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[8] 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.

[9] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java. In *Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications*, pages 312–326, October 2001.

[10] M. J. Harrold and G. Rothermel. A safe, efficient regression testing technique. *ACM Transactions on Software Engineering Methodology*, 6(2), April 1997.

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

[12] 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,url: citeseer.nj.nec.com/kung93class.html.

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

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

[15] A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analysis for java. Technical Report DCS-TR-474, Rutgers University, 2002.

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

[17] 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*, Oct. 2001.

[18] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications*, pages 254–280, October 2000.

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

[20] 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.

[21] 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, 2000.

[22] L. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha. Test manager: a regression testing tool. *International Conference on Software Maintanence*, 1993.