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 ex-
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 depen-
dences. In regression testing imprecision leads to two prob-
lems: (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 de-
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 anal-
yses with different tradeoffs between cost and precision, de-
veloped 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 dem-
strate our approach, but the methodology can be used with
minor modifications with other object-oriented program-
ing languages.

Previous work uses the structure of the program class hi-
ærarchy 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 al-
gorithms presented in [20] and [16]. On a set of nine re-
allistic 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 num-
ber of edges in the ExtORD.

In integration testing, these reductions may result in sub-
stantial savings in time and effort. When using coverage
analysis tools, substantially less time will be spent trying
to exercise impossible interclass dependences. In regres-
sion 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 sig-
ificantly improved precision is beneficial for other uses of the
ORD/ExtORD (e.g., for program understanding and reengi-
neering).

Even though most of our data programs are large, the ex-
perimental 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 con-
struction 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 im-
portance of ORD precision. Section 3 presents a general
algorithm for ORD construction and discusses specific class
analyses. The experimental results are presented in Sec-
tion 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, pro-
gram understanding, and reverse engineering. Once con-
structed, 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) 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(); }
} class C {
X h;
8 C(X xh) { this.h = xh; ... }
   void m() {
9   X xh = this.h;
10  xh.n(); }
} class D {
X i;
11 D(X xi) { this.i = xi; ... }
   void m() {
12   X xi = this.i;
13   xi.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 $(B, I, A)$ 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 $(B, Ag, A)$ if and only if statement $this.f = new A(\ldots)$ appears in a constructor defined in class $B$.¹

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

- $T m(\ldots,A r,\ldots) \{ \ldots \}$ 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 $r.m(\ldots)$ 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.

¹According 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.
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 ((A, As : 4), Y) and ((A, As : 4), 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.x.a at line 2 and A.m.x.a 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.

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, 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

---

2In 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.
of all objects that may be bound to variable \( x \). \( \text{EnCl} \) 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 \( C \in Cs(p) \). 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, \( \text{CHA} \) 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 \((o_1, f)\) 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 \( \text{AND-PT} \) and \( \text{OBJ-PT} \), and their corresponding class analyses, denoted by \( \text{AND} \) and \( \text{OBJ} \). \( \text{AND-PT} \) [20] is a flow-insensitive and context-insensitive points-to analysis and \( \text{OBJ-PT} \) [16] is a context-sensitive version of \( \text{AND-PT} \).

---

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.

---

**Figure 5. ORD construction.**

**Figure 6. Sample program and its points-to graph.**
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 \((o_1, f), o_2\) and \((o_3, f), o_2\). 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 \(s_3: X.p_2 = \text{new} X();\)
6 \(s_4: Z.q_2 = \text{new} Z();\)
7 \(p_2.set(q_2);\)
8 \(Y.q_3 = p_2.f;\)

When these statements are analyzed using AND-PT, there are spurious points-to edges \((o_1, f), o_2\), \((o_3, f), o_2\), 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 \((o_1, f), o_2\), \((o_3, f), o_2\), and \((o_3, o_3)\). 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>User Class</th>
<th>Size (KB)</th>
<th>Whole-program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class</td>
<td>Method</td>
<td>Stmt</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>62</td>
<td>176.8</td>
<td>608</td>
</tr>
<tr>
<td>jtree-1.0</td>
<td>72</td>
<td>272.0</td>
<td>620</td>
</tr>
<tr>
<td>sablecc-2.9</td>
<td>312</td>
<td>532.4</td>
<td>864</td>
</tr>
<tr>
<td>javac</td>
<td>182</td>
<td>614.7</td>
<td>730</td>
</tr>
<tr>
<td>creature</td>
<td>65</td>
<td>259.7</td>
<td>626</td>
</tr>
<tr>
<td>mindterm1.1.5</td>
<td>120</td>
<td>461.1</td>
<td>686</td>
</tr>
<tr>
<td>soot-1.beta.4</td>
<td>677</td>
<td>1070.4</td>
<td>1214</td>
</tr>
<tr>
<td>muffin-0.9.2</td>
<td>245</td>
<td>655.2</td>
<td>824</td>
</tr>
<tr>
<td>javac-1.0</td>
<td>63</td>
<td>502.6</td>
<td>615</td>
</tr>
</tbody>
</table>

Table 1. Characteristics of the data programs.
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. 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.

4.1.3 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.
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


