A Semantics-Based Definition For Interclass Test Dependence

Weilei Zhang, Barbara G Ryder
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
{weileiz,ryder}@cs.rutgers.edu

ABSTRACT

The knowledge of interclass test dependence is important in deciding class test order, facilitating the design of an efficient integration test plan. However to date, interclass test dependence is defined based on a class diagram called the *Object Relation Diagram (ORD)*, which can only capture the syntactical relationships between classes, and introduces spurious dependences not existing in implementation. We explore a semantics-based definition for interclass test dependence. A safe approximation algorithm is designed and implemented to calculate interclass test dependence according to the given definition. The algorithm propagates semantic dependences at method-level granularity and is parameterized by the precision of the corresponding program analysis. We experiment with nine benchmarks and four different analysis configurations. The empirical results show that the algorithm is practical in terms of time cost. On average, the algorithm with the most precise configuration recognizes 75.24% of the dependences from the ORD-based approach as spurious. The algorithm is rather accurate in that it discovers dependence cycles precisely in 6 out of 9 benchmarks (as evaluated by human inspection). The algorithm uncovers additional opportunities for concurrent testing in each of the benchmarks, with an on average gain of 52.5% over the ORD-based definition.

Keywords

Interclass Test Dependence, Class Integration Test, Integration Test Order

1. INTRODUCTION

Object-oriented (OO) programming introduces new challenges for testing because of its language features of encapsulation and polymorphism [4, 28]. Class is the kernel concept in both OO programming and OO testing. A class usually requires other classes to implement its functionalities (e.g., its parent class, a server class, etc). As a result, the execution of a class does not solely depend on its own implementation, but also on the implementations of its prerequisite classes. Intuitively speaking, if class *B*'s implementation affects class *A*'s test result, there is an *interclass test dependence* from *A* to *B*. *A* and *B* are called the *source* and the *target* class of the dependence respectively.[1]

Interclass test dependence is important in deciding class integration test order, facilitating the design of an efficient integration test plan [18]. Such a plan must clarify the test focus for each integration step for the convenience of test coverage evaluation and debugging [4]. Because the source class of a dependence requires the target class to implement its functionalities, it is desirable to integrate and test the source class *after* the target class has been tested. If there is no dependence cycle and the dependence graph is a directed acyclic graph (i.e., DAG), the reverse topological order of the DAG can be used directly as the class integration order. The existence of dependence cycles complicates the problem. There are two approaches to address cycles: (i) to integrate all of the classes in a cycle in one step and test them as a *cluster* that is usually more costly to test and debug than a single class because of the enlarged test scope and broadened test focus [4], or (ii) to break the cycle by constructing test stubs. A test stub is a partial implementation of the target class that simulates the needed functionalities, and is usually difficult to construct [3]. Clearly, obtaining interclass test dependences as accurately as possible may eliminate spurious dependences that contribute to cycles. In the cases where dependence cycles do exist, more precise dependences will keep cycle size as small as possible.

A more precise dependence graph may help to parallelize integration test activities and speed up the test process. The existence of interclass test dependences serializes test activities, since each target class must be tested before its corresponding source class. Classes may be tested concurrently (i.e., in parallel) if there is no dependence between them. This may result in a shorter test process time frame, a very desirable outcome because many software projects are short-changed in terms of the time allowed for testing.

Since interclass test dependence as defined here corresponds to a semantic dependence between classes, it is also useful in many other areas of software engineering. For example in program understanding, knowledge of interclass test dependences can help to prune out unrelated classes, in order to understand better one particular class's behavior. Also, in software visualization, knowledge of dependence may help to organize the class diagram that is especially useful for visualizing a large program.

The current notion of interclass test dependence is defined based on a class model called the *Object Relation Diagram(ORD)*. It was initially defined as the transitive closure of the relationships of inheritance, aggregation and association [18]; later, dependences due

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

[1] In the paper, *dependence* refers to the term *interclass test dependence*, if not specified otherwise.

to polymorphism were added [19]. As a design diagram, the ORD cannot capture the detailed information about the source code. In addition, the ORD-based definition only corresponds to the syntactical relationships between classes, and thus may introduce spurious dependences not existing in the implementation. The goal of the research in this paper is to explore a semantics-based definition for interclass test dependence that improves upon the ORD-based definition when source code (or Java bytecode) is available.

This paper makes the following contributions:

- A new semantics-based definition for interclass test dependence.
- An algorithm to safely approximate the new definition, parameterized by the precision of the corresponding program analysis. Thus the user can control the cost/precision tradeoffs by varying the choice of analysis.
- Experimentation with four versions of the algorithm (using four different analysis configurations) on nine Java benchmarks. The empirical results show that even the most precise version of the algorithm is *practical* in time cost, and *efficient* at reducing the number of spurious dependences, over the ORD-based definition. The algorithm discovers the class dependence cycles accurately in *six out of the seven* benchmarks amenable to manual inspection. The algorithm uncovers additional opportunities for concurrent testing in each of the benchmarks, with an on average gain of 52.5% over the ORD-based definition (see Section 5.2.)

2. ORD-BASED DEPENDENCE

This section introduces the ORD-based definition for interclass test dependence. Then a motivating example shows a spurious dependence present in the ORD-based definition. Starting from the example, a further discussion of spurious dependences is presented.

2.1 ORD-Based Definition

Kung et al. were among the first researchers to address the class integration test order problem [18]. They computed interclass test dependence based on the ORD, an edge-labeled digraph where nodes represent classes, and edges represent the three binary relationships between classes: inheritance, aggregation and association. Figure 1 is a simple example to illustrate the ORD definition. For any two classes X and Y:

- inheritance: an edge labeled I from X to Y indicates that X is a child class of Y.
- aggregation: an edge labeled Ag from X to Y indicates that X is an aggregate class of Y, or in other words, Y is a component class of X (has-a relationship).
- association: an edge labeled As from X to Y indicates that X associates with Y.[2]

Kung et al. define interclass test dependence as the transitive closure of the above three relations. It corresponds to the reachability relation on the ORD (i.e., class X is test dependent on class Y iff there is a directed path from X to Y in the ORD). In Figure 1, an interclass test dependence exists from class D to A and B, but not to C.

Labiche et al. extended Kung's work by considering interclass test dependence caused by polymorphism [19], because a reference variable of a class A may refer to any instance of a subclass of A. As shown in Figure 2, a dashed edge is added from B to C because

[2]The association in the ORD is not exactly the same as the association in UML [40, 41], in which it is a more general concept. The mapping from ORD to UML class diagrams can be found in [6].

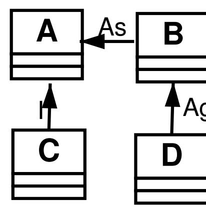


Figure 1: A simple ORD example

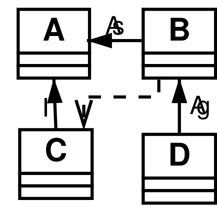


Figure 2: Dependence because of polymorphism

B associates with A and A is the parent class of C. Dependence corresponds to the reachability relation in the new graph. In rest of this paper, the ORD-based definition refers to Labiche's extension unless specified otherwise.

2.2 Motivating Example

As a design diagram, the ORD does not capture the detailed information of the source code; also, the ORD-based definition only corresponds to the syntactical relationship between classes, and may introduce spurious dependences that do not exist in the implementation, as shown in the example in Figure 3.

```
class DeliveryHandler{
  ......
  static private PrintStream outFile;
  public DeliveryHandler(PrintStream ps)
  { synchronized(getClass()) {
      outFile = ps;
    }
  }
  public void handleDelivery
       (DeliveryTransaction deliveryTransaction)
  { deliveryTransaction.process();
    deliveryTransaction.display(outFile);
  }
}
```

Figure 3: A motivating example to show a spurious dependence present in the ORD-based definition of dependence

Figure 3 shows a fragment from the class *DeliveryHandler* in *specjbb* benchmark. As shown, two methods of the class *DeliveryTransaction* are called by *handleDelivery* of *DeliveryHandler*. Some methods of *DeliveryHandler* are also called by *DeliveryTransaction*, which is not shown here. Thus, there are bi-directional association relationships between the two classes. As a result, there is a dependence cycle according to the ORD-based definition; the corresponding integration test plan must deal with this cycle, either by stubbing one of the two classes to test the other first or by testing them together as a cluster.

However, after reading the code in detail it can be seen that the test dependence from *DeliveryHandler* to *DeliveryTransaction* actually does not exist. The behavior of method *handleDelivery* is to call *process()* and then *display(PrintStream)*. The actual parameter passed to *display(PrintStream)* is a static field of *DeliveryHandler*, which is never changed in either of the two callees, so the behavior of method *handleDelivery(DeliveryTransaction)* will not be changed due to the execution of the two methods in *DeliveryTransaction*. Also, *DeliveryTransaction* is not used elsewhere in *DeliveryHandler*, so a true test dependence from *DeliveryHandler* to *DeliveryTransaction* does not exist, and *DeliveryHandler* can be tested before *DeliveryTransaction* without any need for the extra cost of dealing with this cycle.

2.3 Problems with the ORD-Based Definition

Several kinds of spurious dependences found using the ORD-based definition are summarized here.

Case 1: Regarding association and polymorphism.
Suppose there is an association relation from the class *A* to *B* or one of *B*'s predecessor classes, then *A* is considered to be test dependent on *B* according to the ORD-based definition. But in the following cases, a interclass test dependence actually does not exist:

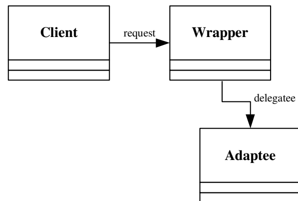
Case 1.1: At runtime, the reference in *A* to *B* or *B*'s predecessor never points to a *B* instance.

Case 1.2: *A* never calls any methods in *B*, although there might be reference from *A* to a *B* instance.
One example for this case is the association from a *Stack* class to its containee. Obviously, the implementation for the containee class is totally transparent to the *Stack* class.

Case 1.3: *A* never calls any methods declared in *B*, although it might call some methods declared in *B*'s predecessor class.
Consider the association from the class *HashMap* to its containee class. *HashMap* calls the *hashCode()* and *equals(Object)* methods of the containee class, but if the containee reuses the two methods inherited from its predecessor class (e.g., *Object* class) instead of overriding them, then *HashMap* is not test dependent on the containee class.

Case 1.4: *A* calls some methods declared in *B*, but those methods have no impact on *A*'s behavior.
Consider the motivating example in Figure 3. The two methods in *DeliveryTransaction* are called, but they neither return any values nor change the value of any variable visible to *DeliveryHandler*. However, this does not mean that the callee methods must be side-effect free. For example, the execution of *process()* in the motivating example might change a memory region read by method *display(PrintStream)*, without causing a dependence from *DeliveryHandler* to *DeliveryTransaction*.

Case 2:Regarding Aggregation.



The ORD-based definition says that aggregation creates a test dependence from the aggregate class to the component class, but this is not always true. Take the scenario shown on the right as an example; it illustrates a *Wrapper* design pattern [1]. Suppose the request from the *Client* is asynchronous and the *Wrapper* is implemented as an aggregate class for the *Adaptee* for better encapsulation. What the *Wrapper* does is just to relay the request to the *Adaptee*, therefore it is not test dependent on the *Adaptee*'s implementation, although there is an aggregate relationship.

Actually, aggregation has been unified into the concept of association as a special case in UML [40, 41]. Compared to the regular association, aggregation defines a transitive and asymmetric relationship among the instances of the classes, but it does not automatically lead to interclass test dependence. Consequently, we do not differentiate aggregation specifically in the new definition and algorithm.

3. SEMANTICS-BASED DEFINITION

We define interclass test dependence based on semantics to improve upon the ORD-based definition when the source code is available. Informally speaking, if the change of one class's implementation may affect another class's test result, there exists an interclass test dependence. Section 3.1 describes how to safely model the class test result, as the basis for the new definition for interclass test dependence presented in Section 3.2.

3.1 A model of Class Test Result

The process of class test is described first, followed by how to evaluate the class test result.

The Process of Class Test. In contrast to testing a single method at a time, class test must exercise the cooperation between all methods in a class.[3] In this respect, class test is like testing an abstract data type with multiple entry points [4]. Figure 4 describes a typical class test process: a sequence of CUT (i.e., the class under test) methods *InwardSeq* (the first of which is a constructor) are exercised. The semantics of the CUT are solely manifest in the return values of CUT methods and side effects on the memory regions outside the CUT instance.

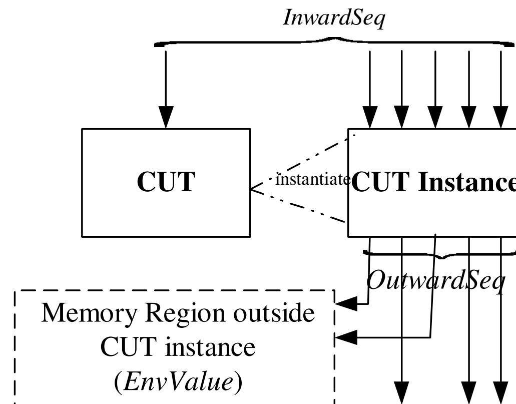


Figure 4: Class Test Process

The following terms are used to describe a class test process:

- *InwardSeq* denotes the sequence of method calls exercised on the CUT during class test.
- *EnvValue* denotes the values for the memory regions outside the CUT instance, and $EnvValue^0$ denotes the initial *EnvValue* before class test.
- *WriteOut* denotes a write from the CUT instance to *EnvValue*. It can be any method call whose side effect changes *EnvValue*.
- *ReturnValue* denotes a value returned by a CUT method.
- *OutStatement* denotes a statement in the CUT producing a *WriteOut* or *ReturnValue*.
- *OutwardSeq* denotes the sequence of *ReturnValue*s and *WriteOut*s triggered by the *InwardSeq*.

How to Evaluate Class Test Result. The class test result is evaluated in order to make sure the CUT is implemented with the expected semantics. As described earlier, the semantics of the CUT is manifest in two ways during class test: *ReturnValue*s and changes to *EnvValue*. However, it is highly impractical to track *EnvValue* extensively for evaluation. We make the following claim:

Claim 1: Given two implementations of a class, if the same *OutwardSeq* is generated for the same ($EnvValue^0$, *InwardSeq*) pair, then the two implementations have the same semantics.

Proof: The claim can be proved by contradiction.
Assume that the above claim is false, (i.e., there are two classes

[3]Without loss of generality, we assume that the access to any field *f* is done via the corresponding methods *setf(value)* and *getf()*.

with different semantics, but the same *OutwardSeq* is generated given the same (*EnvValue*^{0}, *InwardSeq*) pair). Because different semantics means that there is difference in *ReturnValue*s or *EnvValue*, and the same *OutwardSeq* assures that the *ReturnValue*s are the same, thus there is a (*EnvValue*^{0}, *InwardSeq*) pair for which different *EnvValue*s will result. Suppose *ins* is the instruction which causes the first difference in *EnvValue*s. Since *OutwardSeq*s are the same for the two classes, *ins* must be outside the CUT. Because *EnvValue*s before executing *ins* are the same, thus *ins* must be reading from a field *f* with different values in the two class instances. Now the method call *getf()* will return different values, which is contrary to the assumption that *ReturnValue*s are the same for the same (*EnvValue*^{0}, *InwardSeq*) pair, so the claim is proved.

The contrapositive statement of *Claim 1* says that if the CUT is not implemented correctly, a different *OutwardSeq* will be generated for a (*EnvValue*^{0}, *InwardSeq*) pair, so it is safe to evaluate the class test result by observing the correctness of *OutwardSeq* only.

3.2 Semantics-based Formal Definition for Interclass Test Dependence

As shown in Section 3.1, the class test result can be safely evaluated by observing *OutwardSeq*. Thus, we say that there is interclass test dependence from class *A* to *B* if class *B* contains one statement that can affect *A*'s *OutwardSeq*. The definition is formalized as follows:
Definition 1: There is a test dependence from class *A* to *B* if the following is true:

$$\exists sa \in Stmt.\ \exists mb \in Method.\ \exists sb \in Stmt.\ (CdeclM(B, mb) \wedge SinM(sb, mb) \wedge CreachS(A, sb) \wedge CoutS(A, sa) \wedge SdepS(sa, sb, A))$$

where

- *Stmt* represents the set of all statements;
- *Method* represents the set of all methods;
- *Class* represents the set of all classes;
- *CdeclM*(*C:Class*, *m:Method*) represents the predicate that *m* is declared in class *C*;
- *SinM*(*s:Stmt*, *m:Method*) represents the predicate that *m* contains statement *s*;
- *CreachS*(*C:Class*, *s:Stmt*) represents the predicate that *s* may be reachable while testing class *C*;
- *CoutS*(*C:Class*, *s:Stmt*) represents the predicate that *s* is an *OutStatement* in class *C*;
- *SdepS*(*sa:Stmt*, *sb:Stmt*, *C:Class*) represents the predicate that statement *sa* is semantically dependent on *sb* while testing class *C*.

4. APPROXIMATION ALGORITHM

Unfortunately, it is impossible to compute dependence according to *Definition 1*. One obvious problem is that the semantic dependence predicate (*SdepS*) is undecidable. Podgurski and Clarke present the idea of *weak syntactic dependence* as a safe approximation for semantic dependence [29]; however, it is extremely costly to compute *weak syntactic dependence* between statements. In order to have a practical implementation, we approximate semantic dependence at method-level granularity, and design the approximation algorithm accordingly.

4.1 Approximate Definition

A flow-insensitive approximation is used: inside a method, the relative position of statements is ignored and each use is considered to be semantically dependent on any definitions in the same method. If some statement inside method *ma* is semantically dependent on some statement in method *mb*, we say that any statement in *ma* is semantically dependent on any statement in *mb*. Therefore, semantic dependence can be propagated at method-level granularity on the call graph. Obviously, the precision of the call graph influences the computation for interclass test dependence.

Conservatively assuming each method in the call graph may affect the *OutwardSeq* of its class, interclass test dependence can be safely approximated as follows:
Definition 2: Class *A* is considered to be test dependent on *B* according to our approximation if the following is true:

$$\exists ma \in Method\ .\ \exists mb \in Method.\ (CdeclM(B, mb) \wedge CreachM(A, mb) \wedge CentryM(A, ma) \wedge MdepM(ma, mb, A))$$

where

- *CdeclM* and *Method* are the same as in *Definition 1*;
- *CentryM*(*C:Class*, *m:Method*) represents the predicate that *m* is a method exercisable on a *C* instance. *m* can be declared in *C* or one of *C*'s predecessor classes;
- *CreachM*(*C:Class*, *m:Method*) represents the predicate that method *m* is reachable from class *C* (i.e., *m* may be called while testing *C*);
- *MdepM*(*ma:Method*, *mb:Method*, *C:Class*) represents the predicate that while testing class *C*, method *ma* is considered to be semantically dependent on method *mb* according to our approximation. Although it is an approximation, we refer to it as *semantic dependence* for brevity in later discussions.

Examining *Definition 2*, we can introduce the predicate *CdepM*:

- $CdepM(A, mb) \equiv \exists ma \in Method.(CreachM(A, mb) \wedge CentryM(A, ma) \wedge MdepM(ma, mb, A))$.

Informally speaking, *CdepM*(*C:Class*, *m:Method*) means that class *C* is dependent on method *m* (i.e., one method declared in *C* is semantically dependent on *m* while testing *C*). Using *CdepM*, *Definition 2* can be rewritten as follows:
Definition 3: Class *A* is considered to be test dependent on *B* according to our approximation if $\exists mb \in Method.(CdeclM(B, mb) \wedge CdepM(A, mb))$.

4.2 Calculating Interclass Test Dependence

As shown in *Definition 3*, the main step for the approximation algorithm calculating interclass test dependence is to compute *CdepM*. Figure 5 shows the constraints used to compute *CdepM* in Datalog [17],
where

- *CentryM*, *CreachM* and *CdepM* are defined previously.
- *call*(m:Method, i:Method)* represents the predicate that method *m* calls *i* directly or indirectly.
- *MretM(m:Method, i:Method)* represents the predicate that method *i* calls *m* directly and reads the return value of *m*.
- *Mread, Mwrite(m:Method,o:AbstractNode,f:Field)* represent the predicates that *m* may read from or write to the abstract memory region *o.f*, respectively. The two predicates are used to calculate inter-method semantic dependence due to side effects (See below for more details).

```
CentryM(c:Class, m:Method) input
CreachM(c:Class, m:Method) input
call*(m:Method, i:Method) input
MretM(m:Method, i:Method) input
Mread(m:Method, o: AbstractNode, f:Field) input
Mwrite(m:Method, o: AbstractNode, f:Field) input
CdepseM(c:Class, m:Method)
CdepM(c:Class, m:Method) output

(0)CdepM(c,m):- CentryM(c,m).
(1)CdepM(c,m):- CentryM(c,i),CreachM(c,m),call*(m,i).
(2)CdepM(c,m):- CdepM(c,i),MretM(m,i).
(3.1)CdepseM(c,m):- CdepM(c,i),CreachM(c,m),
                    Mread(i,o,f),Mwrite(m,o,f).
(3.2)CdepM(c,m):- CdepseM(c,m).
(3.3)CdepM(c,m):- CdepseM(c,i),CreachM(c,m),call*(m,i).
```

Figure 5: A Datalog Program to Compute CdepM

- *CdepseM(C:Class, m:Method)* represents the predicate that class *C* is dependent on method *m* due to side effects (See below for more details).

A class is dependent on all the methods exercisable on its instances, so *CdepM* is initially equal to *CentryM*, as shown in rule 0. In rules 1-3, the dependences in *CdepM* propagate to other methods because of the existence of inter-method semantic dependences. There are three causes for inter-method semantic dependence, one due to side effects and the other two due to call relations: from the caller to the callee and from the callee to the caller.

Rule 1: From Callee To Caller. The callee is considered to be semantically dependent on the caller, because it is control dependent on the call site in the caller. Consequently, *CdepM* also contains those methods which may call *C*'s entry method(s) and are reachable from *C*.

Rule 2: From Caller To Callee. Because Java is *call by value*, ignoring the existence of side effects the caller is considered to be semantically dependent on the callee only if it reads the value returned by the callee (i.e., *MretM(callee,caller)*). If side effects exist, rules 3.1-3.3 are applied.

Rules 3.1-3.3: Side Effects. There are inter-method semantic dependences due to side effects because different methods may access the same memory region on the heap (i.e., the same object fields, array items or global variables).[4] In our approximation algorithm, method *ma* is considered to be semantically dependent on *mb* if *ma* reads from a memory region that *mb* may write to. This is a flow-insensitive approximation because it disregards the order of the reads and writes. As shown in rule 3.1, if method *i* is semantically dependent on *m* due to side effects, and class *C* can reach *m*, then the dependence of *C* on *i* results in a dependence of *C* on *m*.

The dependence of class *C* on method *m* due to side effects is stored in *CdepseM* first, and propagated to *CdepM* at rule 3.2. If *C* is dependent on *m* due to side effects, *C* is also dependent on those methods reachable from *C* that call *m* directly or indirectly. Therefore, we use *CdepseM* to calculate these dependences, as shown in rule 3.3.

Class *C* is also dependent on method *m* if *m* is reachable from *C* and changes one of *C*'s static fields or a *C* object's fields. This is handled in the same way as side effects. The rule is not shown here for brevity.

[4]There is also a semantic dependence from method *ma* to *mb* if *ma* calls *mb* directly or indirectly and *mb* changes the program flow viewable by *ma* (e.g. *mb* calls *System.exit(int)* or throws an exception viewable by *ma*). This semantic dependence is modelled and dealt with in a similar way as that due to side effects.

Algorithm. The algorithm is parameterized by the program analysis used to calculate the call graph and side effects. More details on call graph precision are presented in Section 4.3. Points-to analysis approximates the memory regions on the heap a method accesses (*Mread*, *Mwrite*) in order to evaluate the inter-method semantic dependences caused by side effects. A memory region is represented as a *(AbstractNode, Field)* pair. *AbstractNode* is determined by the results of points-to analysis; its representation can be, but is not limited to, an allocation site, a class or an array. *Field* corresponds to the field of the class or the index of the array, which can be ignored if field-insensitive analysis is used to propagate semantic dependence. In our implementation, because of the lack of an appropriate static analysis algorithm, the array index is always ignored and all items of an array are regarded as a single unit.

The rules in Figure 5 define the problem by a set of recursive equations. In Datalog, a fixed-point iteration is used to solve the set of rules (e.g., a change to *CdepM* at rules 2, 3.2 or 3.3 causes a re-evaluation of rules 2 and 3.1; similarly, a change to *CdepseM* at rule 3.1 causes re-evaluation of rules 3.2 and 3.3). Rules are applied until a fixed point is reached. The final result for *CdepM* is used to calculate interclass test dependence according to *Definition 3*.

Note that inter-method semantic dependence is not simply a transitive closure of the relations expressed in these rules. In Figure 6, methods *ma* and *mb* both call *mc*. *ma* reads the value returned by *mc* and thus is considered to be semantically dependent on *mc*. *mb* calls *mc* and thus *mc* is considered to be semantically dependent on *mb*. Barring the existence of any other relations (i.e., call or side effects) among the three methods, obviously no semantic dependence exists from *ma* to *mb*.

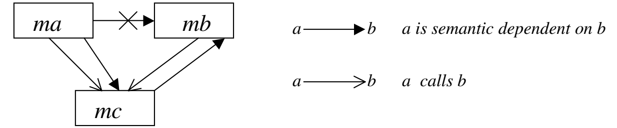


Figure 6: An example

4.3 Precision Of Call Graph

The call graph determines the precision of many predicates used in Figure 5 (e.g., *CreachM*, *Call**). We implemented an object-sensitive points-to analysis [25] to construct an object-sensitive call graph. We further improved the precision of call graph by implementing a data reachability analysis [9] to resolve library callbacks.

Context-Insensitive vs. Object-Sensitive Call Graph. In previous work, *0-CFA* [33, 30, 21] was the most precise points-to analysis we used to compute interclass test dependence [45]. *0-CFA* generates a context-insensitive call graph, in which there is at most one node for each method. Use of *0-CFA* results in many spurious interclass test dependences, as illustrated in the example from the *specjbb* benchmark shown in Figure 7.

AsciiMetrics is a subclass of *Metrics* and overrides method *wrap (String)*. Method *output_properly(String)* is invoked both on *Metrics* and *AsciiMetrics* instances, so in a context-insensitive call graph both *AsciiMetrics:wrap (String)* and *Metrics:wrap (String)* appears as its callee, and the predicate *CreachM(Metrics, AsciiMetrics: wrap (String))* will be true, as shown in Figure 8. Also because *output_properly(String)* calls and reads the return value of *wrap*, there will be interclass test dependence from *Metrics* to *AsciiMetrics* according to the rules 0 and 2 in Figure 5.

Obviously, *output_properly(String)* invoked on any *Metrics* in-

```
class Metrics{
  public void output_properly(String s)
  {buf.append(this.wrap(s));  ... ...  }
  public String wrap(String s)
  { return ("<h1>" + s + "</h1>"); }
}
public class AsciiMetrics extends Metrics{
  public String wrap(String s)
  { return s + "\n";}
}
```

Figure 7: Imprecision of context-insensitive call graph

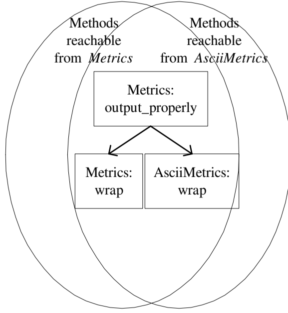


Figure 8: Context-Insensitive Call Graph

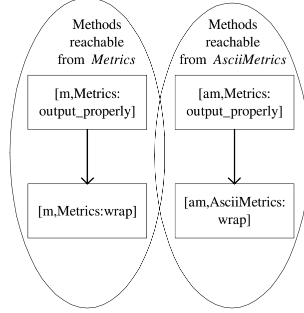


Figure 9: Object-Sensitive Call Graph

stance will call *Metrics:wrap (String)* instead of *AsciiMetrics:wrap (String)*, and in this case, the corresponding interclass test dependence from *Metrics* to *AsciiMetrics* is spurious. We used object-sensitive points-to analysis to eliminate this spurious dependence.

The key idea of object-sensitive points-to analysis is to analyze a method separately for each of the object names that represent different run-time objects on which this method may be invoked. In our implementation, an object is named by its allocation site. In the generated object-sensitive call graph, each method may be represented by multiple nodes with different allocation sites as its calling contexts (i.e., different receiver objects). Each node has the form of *[allocation site, method signature]*. Back to the example in Figure 7, suppose *m* and *am* are two allocation sites with type *Metrics* and *AsciiMetrics* respectively, and *output_properly (String)* is invoked on both of them. The corresponding object-sensitive call graph is shown in Figure 9, from which we can see that the predicate *CreachM (Metrics,AsciiMetrics:wrap (String))* will be false, and there will not be an interclass test dependence from *Metrics* to *AsciiMetrics*.

The programming idiom here is very typical in OO applications. In our experimental findings, after applying object-sensitive points-to analysis, the class test dependence cycles can be determined accurately in four more benchmarks than with the context-insensitive analysis. There are other call-site-based context-sensitive points-to analyses such as *k-CFA* and *cloning-based* method [10, 31, 43]; however, because of the existence of truly polymorphic call sites, the inaccuracy encountered in this example cannot be fully resolved by those analyses without using object sensitivity.

Data Reachability Analysis Resolve Library Call-backs Although we are only interested in dependences between the application classes, the effects of the Java library have to be considered: (i) two application methods calling related library methods may be semantically dependent on each other due to side effects in the library; (ii) one application method calling a library method may reach and semantically depend on another application method because of library call-backs. The library call-backs may introduce inaccuracy because in both the context-insensitive and object-sensitive call graphs, information from different call sites may be merged in the Java library. For example in Figure 10, at runtime method *X:appendA()* only reaches *A:toString()* through the library. In a context-insensitive call graph, *String:valueOf(Object)* calls both *toString()* methods; consequently, *X:appendA()* is seen to reach *B:toString()* mistakenly. This inaccuracy also exists in object-sensitive call graph because *String: valueOf(Object)* is a *static* method, and it cannot be analyzed separately according to different objects as calling contexts. [5]

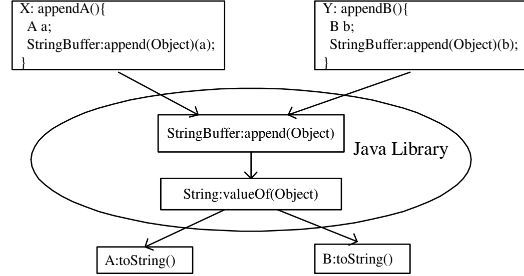


Figure 10: Call Graph For StringBuffer:append(Object)

We solved this problem using data reachability analysis [9]. Data reachability analysis calculates the methods reachable from a call site. We designed and implemented a variation of *V-DataReach* presented in [9]. In our implementation, data reachability analysis has access to the result of a (global) points-to analysis. At each call site from an application method calling a library method, a *call-site-specific* points-to analysis is performed. In this analysis, we calculate a set of *local* objects that are created during the call and destroyed after the call returns. As with any data reachability analysis, a sub-call graph reachable from the call site is constructed on the fly. The *local* objects' points-to information is computed by a points-to analysis that uses the sub-call graph under construction; (non-*local*) objects' points-to information comes from the global points-to analysis. Eventually, data reachability analysis calculates the possible call-backs to application methods for each site calling a library method.

5. EXPERIMENTS

We implemented an object-sensitive points-to analysis, a data reachability analysis and the algorithm to calculate interclass test dependence. We experimented on nine benchmarks, including all eight benchmarks from SPECjvm98 [6] and SPECjbb2000.[7] All experiments were run on 2.8GHz Pentium-IV, 1.5GB-memory PC with Linux 2.4.20-13.9 and SUN JVM 1.4.1_03-b02.

5.1 Experimental Setup

The implementation framework has two separate modules, as shown in Figure 11. The first module uses the Java analysis and transformation framework, *Soot* [32] version 2.2.1; the second uses a *BDD* based solver, *bddbddb* [43, 16]. There is also some post-processing for evaluation.

[5] A variation of the object-sensitive analysis can actually solve this particular problem: a static method is analyzed separately according to each of its caller's calling context(s). But this variation will slow down the object-sensitive analysis greatly, and cannot resolve the inaccuracy as generally as data reachability analysis.

[6] http://www.spec.org/jvm98/

[7] http://www.spec.org/jbb2000/

The task of the first module is to generate constraints using the results of some fundamental program analyses. The constraints are

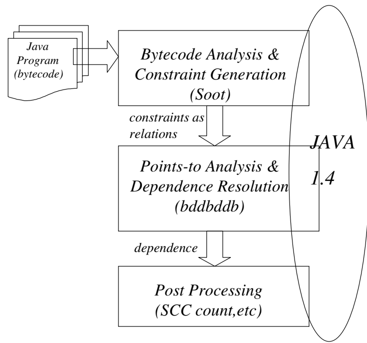


Figure 11: Framework

represented as relations, and basically include the following information:

- Static information for the class: its inheritance relations; the fields and methods it declares, etc.
- Pointer assignment edges: assignment, store, load and allocation edges.
- Side-effect information: the fields of reference variables and the global variables each method reads from or writes to.
- The call relation between methods: the *CHA* (class hierarchy analysis) call graph [8], and whether the caller ever reads and uses the return value of callee.

Based on the constraints, the second module performs the analyses (points-to analysis and data reachability analysis) to generate the call graph and the concrete side-effect information: the memory regions on the heap each method may access (read or write). Then, interclass test dependence is calculated.

The framework is parameterized by the choice of analyses, and different analyses can be applied to balance the trade-off between efficiency and precision. In our experiments, we use the following four analysis configurations, in order of increasing precision:

- *VTA*: variable type analysis. The target nodes in the pointer assignment graph are class types, not allocation sites [35]. In *Soot*, the call graph is not constructed on the fly for *VTA*;
- *0CFA*: a field-sensitive, flow-insensitive and context-insensitive points-to analysis (a form of *0-CFA*) [33, 30, 21]. In *Soot*, the call graph is constructed on the fly for *0CFA*;
- *OB*: a field-sensitive, flow-insensitive and context-sensitive points-to analysis. Each method is analyzed separately for the object names on which this method is invoked [25], and each object is named by its allocation site;
- *OBR*: object-sensitive points-to analysis is used to construct the initial call graph and calculate side effects. Then, the data reachability analysis [9] presented in Section 4.3 is applied in order to resolve library call-backs. The points-to results of the object-sensitive analysis are also used as a filter during the *call-site-specific* points-to analysis.

5.2 Results

Three sets of questions need to be answered in order to evaluate the results for the semantics-based definition computed:

- How many spurious dependences are eliminated according to the given algorithm, and how do these results vary with different analysis configurations?
- How do the results vary in terms of cycles in the dependence graph? This question is important because it is dependence cycles that complicate the issues of determining class test order and designing an efficient integration test plan.
- How can we effectively evaluate the improvement in the integration test plan?

The experiments were designed to answer the above questions and the corresponding findings are shown in this section.

Spurious Dependences Figure 12 shows the normalized data for the number of computed interclass test dependences. The y-axis is the percentage of the number of dependences computed according to the given algorithm, divided by the number computed according to the ORD-based definition. As expected, the *VTA* configuration results in the largest number of dependences (on average 56.70% of the dependences in the ORD-based definition), i.e., the least number of spurious dependences eliminated. *0CFA*, *OB* and *OBR* amount to, on average, 32.17%, 25.09% and 24.46%, respectively.

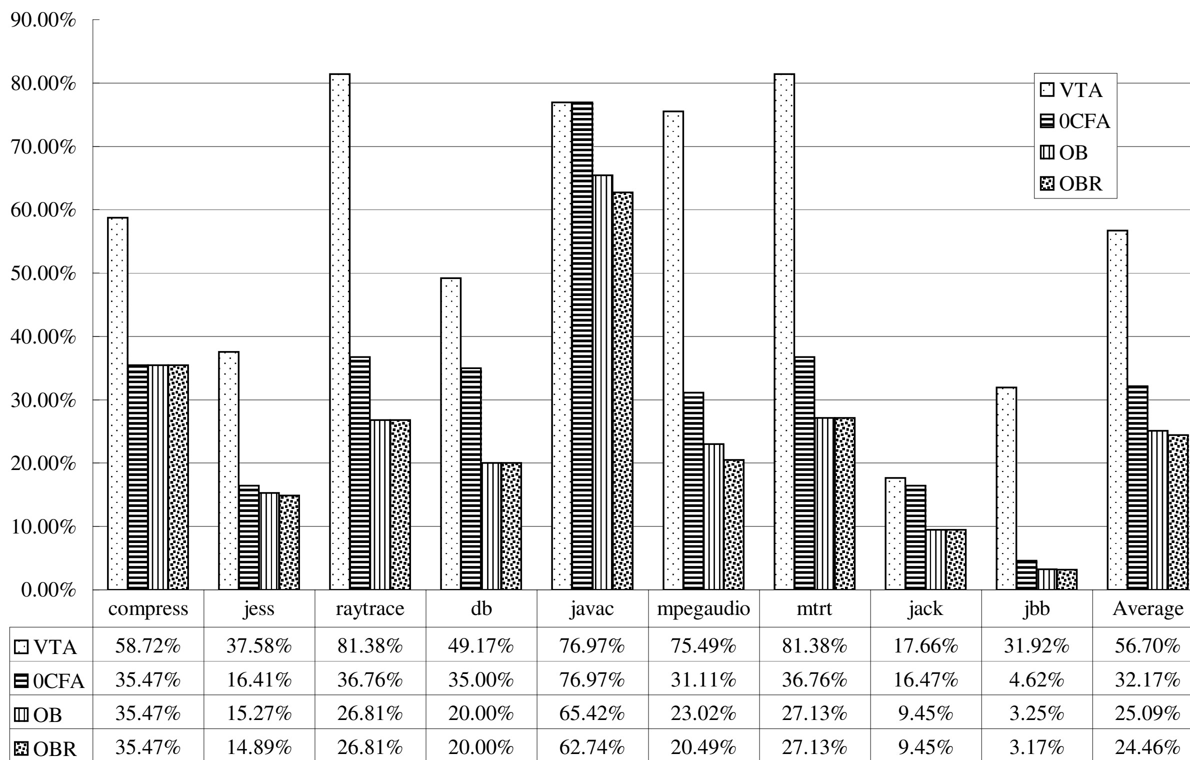
Dependence Cycles Table 1 shows the size of each cycle in the test dependence graph. Each box contains a sequence of decreasing numbers that correspond to the size of all cycles for each benchmark computed with a particular algorithm configuration. We also computed dependence cycles according to the ORD-based definition and the results are shown in column *ORD*. If the computed result is accurate according to manual inspection, the number is shown in underlined bold font. The number of total application classes is also shown in parentheses next to the benchmark name. Taking the *raytrace* benchmark as an example, it has 41 application classes. There are three dependence cycles according the ORD-based definition, containing 18, 4 and 2 classes respectively. According to the given algorithm with the *VTA* configuration, the 4-class and 2-class cycles do not exist, and the 18-class cycle shrinks to the size of 16. According to the *0CFA* configuration, the 16-class cycle breaks into one 8-class cycle, one 4-class cycle and with the four remaining classes not in any cycle. According to the *OB* and *OBR* configurations, the 8-class cycle further shrinks to the size of 7, and the 4-class cycle no longer exists, which is accurate by manual inspection.

	ORD	VTA	0CFA	OB	OBR
compress(28)	6,4,4	5,3	**3**	**3**	**3**
jess(163)	147,4	139	96	94	90
raytrace(41)	18,4,2	16	8,4	**7**	**7**
db(20)	10	6	5	**0**	**0**
javac(184)	169	161	161	142	134
mpegaudio(60)	44	37	6,3,2	**0**	**0**
mtrt(41)	18,4,2	16	8,4	**7**	**7**
jack(69)	44	7,6,2,2	7,4,2,2	7	7
jbb(104)	79	49	3,3,2	2,2	**2**

Table 1: Size of Dependence cycles (bold means accurate result)

The source code for *javac* and *jack* is not available, so we could not manually determine the accurate solutions. Out of the other seven benchmarks, we found that the *OBR* configuration determines the classes in the cycle accurately in six benchmarks (i.e., all except *jess*).

As for the *jess* benchmark, a lot of inaccuracies were found to result from the use of *java.util.Hashtable*. *Hashtable* elements are internally stored in an array. In our implementation, an object is named by its allocation site. Thus, different *Hashtable* instances



	compress	jess	raytrace	db	javac	mpegaudio	mtrt	jack	jbb	Average
VTA	58.72%	37.58%	81.38%	49.17%	76.97%	75.49%	81.38%	17.66%	31.92%	56.70%
0CFA	35.47%	16.41%	36.76%	35.00%	76.97%	31.11%	36.76%	16.47%	4.62%	32.17%
OB	35.47%	15.27%	26.81%	20.00%	65.42%	23.02%	27.13%	9.45%	3.25%	25.09%
OBR	35.47%	14.89%	26.81%	20.00%	62.74%	20.49%	27.13%	9.45%	3.17%	24.46%

Figure 12: Number of Interclass Test Dependences: **y-axis is the percentage of #(dependences according to the given algorithm) divided by #(dependences according to the ORD definition)**

share the same allocation site for the array, and all the information passing through it will be merged. The presented data reachability analysis cannot solve the problem either, because the array is not a *local* object for the *call-site-specific* points-to analysis, and its points-to information comes from the global points-to analysis. A more precise naming scheme for objects (e.g., a context-sensitive naming scheme to distinguish the different objects created at the same allocation site) may solve the problem.

Comparing Table 1 to Figure 12, the improvement from *0CFA* to *OB* is modest in the number of dependences, but it is interesting to see that *OB* accurately can find cycles in five benchmarks, while *0CFA* does so in only one. This is mainly because a context-insensitive call graph may cause spurious dependences between closely related classes, and elimination of those spurious dependences usually causes cycles to shrink. Recall the example in Figure 7, class *AsciiMetrics* is a subclass of *Metrics* and overrides a method in *Metrics*, causing a spurious test dependence from the parent class (e.g., *Metrics*) to the child class (e.g., *AsciiMetrics*). The child is often found to be dependent on the parent, so deletion of the spurious dependence from the parent to the child by using an object-sensitive call graph usually breaks the cycle and improves precision.

Maximum Path Weight By eliminating spurious dependences, more classes may be tested in parallel to speed up the test process. Testing is often done on a very tight timetable, so we estimated the minimum time span needed for class integration test to evaluate the possible improvement in the test plan. We used the following considerations and assumptions:

- If a dependence cycle exists, the classes in the cycle are integrated together in one step and tested as a cluster. As mentioned in Section 1, there are two approaches to handle dependence cycles, one is to test classes in a cycle as a cluster, and the other is to break cycles by constructing test stubs. We assume the former occurs because usually it is preferable not to construct test stubs to simulate classes that are already implemented.
- For simplification, we assume a uniform time cost for each class, denoted as one unit of cost.
- Based on the consideration that the time cost to test all classes in a cycle (i.e., cluster test) increases as the size of the cycle increases, we assume that the cost is a linear function of the size[8], i.e., it cost n units to test a n-class D cycle. Conse-

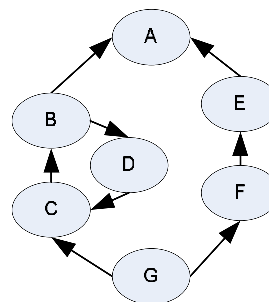


Figure 13: Class Dependence Graph

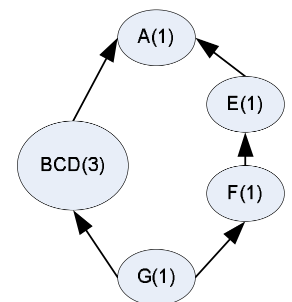


Figure 14: DAG with Weighted Node

[8] Actually,the cost to test a cycle usually increases more quickly than the size does because of the broadened test focus; however, the linear assumption, albeit simple, is safe and good enough to illustrate the improvement in the integration test plan.

quently, given a class dependence graph, by condensing each strongly connected component into a single node, the time cost for testing each class and cluster can be shown in a DAG with weighted nodes. Take Figure 13 as an example. It is a 7-class dependence graph, with each node labeled by the class name. Figure 14 is the corresponding weighted DAG. Each node corresponds to a class or a cycle and its weight is shown in the parenthesis (e.g., node *BCD* corresponds to the cycle containing classes *B*, *C* and *D*, and its weight is three).

- A class or cluster can be tested when all classes and clusters it is dependent on have been tested.
- Different classes and clusters can be tested in parallel if there is no dependence between them so that the time span for integration test can be reduced. Of course, it requires that there are sufficient resources to do so. For example in Figure 14, cycle *BCD* and class *E* can be tested in parallel after testing *A*. According to our assumptions, testing *E* finishes earlier than testing cycle *BCD*, then *F* will be tested in parallel with the cycle.

	ORD		OBR		
	Max Path	Unit	Max Path	Unit	Gain
compress	(6)+(4)+(4)	14	3+(3)	6	57.1%
jess	(147)+(4)	151	4+(90)	94	37.7%
raytrace	1+(18)+(4)	23	6+(7)	13	43.5%
db	1+(10)	11	6	6	45.5%
javac	1+(169)	170	4+(134)	138	18.8%
mpegaudio	2+(44)	46	17	17	63.0%
mtrt	1+(18)+(4)	23	6+(7)	13	43.5%
jack	2+(44)	46	6+(7)	13	71.7%
jbb	3+(79)	82	7+(2)	9	89.0 %
				average	52.2 %

Table 2: Maximum Weighted Path

Based on the above assumptions, the maximum path weight in the DAG corresponds to the minimum time span needed for the class integration test. We used this metric to illustrate the possible improvement in the test plan. In Table 2, the maximum path weight for interclass test dependence computed with the OBR configuration is compared against that with the ORD-based definition. Column *Max Path* describes the maximum weighted path; the number in the parentheses represents the size of cycle on the path. Column *Unit* is the total weight for the path in units. For example, in Figure 14, the maximum weighted path is *A-BCD-G*, so it will be described as 2+(3), meaning there are two single classes and a 3-class cycle on the path, whose weight is 5 units. Column *Gain* illustrates the possible improvement in the class integration test plan in terms of minimum time span needed, that is, the percentage of the difference between the two path weights divided by the maximum path weight in the ORD-based graph. The average improvement is 52.2%.

5.3 Time Costs

Table 3 shows the time costs for the analyses on all benchmarks. Points-to analyses for *VTA* and *0CFA* were based on *Soot*; all the other analyses were implemented and run in *bddbddb*. We also implemented a *0CFA* in *bddbddb*, and its time cost was similar to the *Soot* implementation.

Our analysis always finishes in less than 35 minutes. We believe that the cost of automatic dependence analysis can be justified by the resulting improvement in the integration test plan. For example, there is a 44-class cycle in the *mpegaudio* benchmark according to the ORD-based definition. After running our analysis with the *OB* or *OBR* configuration in minutes, the whole cycle is found to be spurious. Thus, there will be no extra cost to handle that cycle. In the improved test plan, the reverse topological order in the dependence graph generated by the *OB* or *OBR* configuration can be used directly as the class test order. The analysis is efficient, in that the average time our analysis takes to decrease the minimum time span needed by one unit is less than 50 seconds in the worst case (*javac*).

More Details. In Java, *java.lang.StringBuffer* is used widely to handle most *String* operations, and all information passing through it will be merged in a context-insensitive analysis, as illustrated in Figure 10. Consequently, we chose to group all *StringBuffer* allocation sites as a single object in *0CFA*, because it decreased the computation time without decreasing the precision. This inaccuracy is resolved in *OBR*, so the grouping was not performed.

The points-to analysis in *OB* and *OBR* is the same object-sensitive analysis. It constructs an object-sensitive call graph on the fly and costs more than *0CFA*. The order of magnitude higher cost for object-sensitive points-to analysis is justified. because *OB* determines cycles accurately in four more benchmarks than *0CFA*, Data reachability analysis is relatively time consuming, too. Its time cost is related to the number of calls to the Java library. For example, *jbb* has the most calls to the library, 2513, while *compress* has the least, 822, and the corresponding time costs for the two benchmarks' data reachability analysis are the largest and the smallest, respectively. In the current implementation, all calls to the library are analyzed. As future work, we will try to improve efficiency by only analyzing suspicious calls (e.g., the calls may cause library call-backs).

The computation to solve for interclass test dependence is rather inexpensive. Basically, the more precise the points-to analyses, the less time the dependence resolution process takes, because the call graph is smaller and more precise.

6. RELATED WORK

6.1 Class Integration Test & Interclass Test Dependence

There have been many algorithms on computing optimal class test order based on interclass test dependence in order for an efficient integration test plan [18, 19, 36, 39, 7, 6, 5, 12]. The algorithms were mainly designed to reduce the number of constructed test stubs in order to resolve the complications introduced by dependence cycles. They differ in how to choose dependences to eliminate to break cycles.

Bottom-up integration is assumed by all the above papers and also this paper: the target class/cluster of the dependence is always tested before the source. Conversely, top-down integration occurs when an early test on the source class/cluster is performed, but the target has not yet been implemented [26]. Top-down integration incurs a greater cost, to construct test stubs, so it is usually not preferred, especially when an implementation for the target is available. As for cluster test, Big Bang and Backbone integration [2] can be used to test closely related components on dependence cycles. It is beyond the scope of this paper to discuss in detail how to construct test suites for clusters.

To date, all the approaches for class integration test used similar ORD-based definitions for interclass test dependence. Our work can greatly improve test plan design and test ordering algorithms by pruning out spurious dependences, as shown in our experiments.

Our work is closely related to Milanova's and Ryder's [24], which used points-to analysis to generate *ExtORD*, an extended ORD. The ExtORD incorporates polymorphism into the ORD by adding class

	VTA		0CFA		OB		OBR		
	points-to	dependence	points-to	dependence	points-to	dependence	points-to	reachability	dependence
compress	203.6	57.4	86.6	15.1	359.7	9.0	359.7	455.3	3.5
jess	242.5	107.9	88.8	25.4	1176.4	15.3	1176.4	536.5	14.9
raytrace	225.3	55.0	80.8	19.3	371.1	9.6	371.1	509.7	4.9
db	208.4	56.1	79.5	16.9	357.8	13.6	357.8	476.0	5.3
javac	267.4	40.0	97.6	49.9	696.1	25.5	696.1	836.4	29.3
mpegaudio	223.1	59.4	103.5	22.2	810.7	11.6	810.7	477.5	6.9
mtrt	210.4	53.9	94.8	19.1	366.4	14.1	366.4	497.5	5.5
jack	289.8	72.7	93.6	21.6	442.5	12.1	442.5	568.6	11.9
jbb	242.2	82.7	96.0	38.9	445.9	16.4	445.9	1512.6	25.7

Table 3: Time Costs (in seconds)

associations as calculated by points-to analysis. Using the ExtORD rather than the ORD as a basis for a definition of dependence improves precision due to the refined binary relationships calculated. The main difference with our work is that an ExtORD-based definition of class test dependence does not capture the same semantic dependences as does our definition. As illustrated in Figure 3, interclass test dependence is not necessarily implied by the binary relationships in the ORD or ExtORD. An ExtORD-based definition of class dependence cannot eliminate the spurious dependences in Cases 1.2, 1.3, 1.4 and 2 summarized in Section 2.3. Reference [24] used the reduction rate in average *class firewall* ([18]) size to measure improvement over the ORD. A class firewall for Class A is a set that contains all classes on which A is test dependent. To compare with the ExtORD results, we calculated the reduction rate for the two common benchmarks, *javac* and *mpegaudio*. The reduction rates are 15.1% and 42.1% in [24], while our corresponding rates are 23.0% and 67.0% for the *0CFA* configuration and 37.3% and 78.3% for the best configuration (*OBR*). We used the ORD-based definition as the baseline for comparison, because it has been widely used in class integration test. As future work, we will empirically investigate our improvement over [24] on more benchmarks.

6.2 Program Dependence & Slicing

Podgurski and Clarke presented a formal, general model of program dependence[29]. Two generalizations for control and data flow dependence, called *weak* and *strong* syntactic dependence, were introduced and related to the semantic dependence between statements. *Weak* syntactic dependence was shown to be a necessary condition for semantic dependence.

Program Slicing. A program *slice* consists of a set of program statements that potentially affect the values computed at some point of interest, referred to as a *slicing criterion*. Program slicing was originally presented by Weiser in 1979 [42]. Several approaches have been proposed to compute the slices statically [37, 44]. Ottenstein and Ottenstein restated the problem in terms of a reachability problem on a *program dependence graph* [27]. Horwitz, Reps and Binkley introduced the *SDG* (system dependence graph) for inter-procedural slicing [15], and designed a two-pass traversal algorithm. *SDG* has been widely used in slicing and improved in various aspects: to represent arbitrary inter-procedural control flow [34]; to handle multi-threaded programs [14]; to improve the precision on programs with pointers using equivalence analysis [23], etc.

OO Slicing. Larsen and Harrold initially extended the *SDG* to support object-oriented programs [20]; the representation was improved by Liang and Harrold in terms of precision and efficiency [22]. Tonella et al. introduced the results of points-to analysis to help dependence analysis handle pointers and polymorphic calls [38]. Hammer and Snelting implemented an improved slicer to handle the case in which nested objects are used as actual parameters [11].

Issue of Practicality. One major difference between the above static slicing algorithms and the semantic dependence computation in our algorithm is the issue of practicality. They computed semantic dependence at statement-level granularity, whereas, we compute semantic dependence at method-level granularity. The cost for the statement-level approaches is substantially higher in terms of both time and space. In addition, since we analyze the Java library, method-level granularity seems more practical. Considering the Java 1.4 library, even the smallest benchmark in our experiments has more than 600 classes and 3000 methods in the *0-CFA* call graph. To the best of our knowledge, there has not been an efficient slicing implementation for a whole Java program with the Java 1.4 or newer library. Also, we used the object-sensitive call graph to improve our analysis; there has not been an efficient slicing implementation using a context-sensitive call graph.

Our approach to trade off precision for practicality appears to be feasible from our empirical results. Also, because of the flow-insensitive approximation, our approach can be directly applied to multi-threaded programs, while many slicing algorithms cannot.

6.3 Points-to & Data Reachability Analysis

There is a wide variety of reference and points-to analyses which differ in terms of cost and precision. An in-depth discussion on the dimensions of reference and points-to analysis can be found in [31, 13]. In our current experiment, we mainly used variable type analysis [35], a form of 0-CFA [33, 30, 21] and an object-sensitive points-to analysis [25].

Data reachability analysis calculates the methods or call chains reachable from a call site. Reference [9] presented a detailed discussion on data reachability analysis for Java. We designed and implemented a variation of *V-DataReach*. The improvement in our approach is that we calculated the set of *local* objects for each call site, whose points-to information comes from a *call-site-specific* points-to analysis.

7. CONCLUSION

We have presented a new semantics-based definition for interclass test dependence. We have designed and implemented a safe approximate algorithm to propagate semantic dependence at method-level granularity. We have experimented with four analysis configurations and nine benchmarks. The empirical results show that the algorithm is not only practical, but also rather accurate. In three benchmarks, the algorithm with the most precise configuration improves over the accuracy of the ORD-based dependence by an order of magnitude, and determines the classes in cycles accurately in *six out of the seven* benchmarks amenable for manual inspection. Also,

the algorithm uncovers opportunities for concurrent testing. Overall, by using our analysis a more efficient class integration test plan can be achieved.

Acknowledgments. We would like to thank Chen Fu to offer insightful comments on the work, and suggest using data reachability analysis to improve the precision. We are also thankful for Bruno Dufour and other Prolangs members' valuable feedback, and for John Whaley's timely help on using *bddbddb*.

8. REFERENCES

[1] F. A. Arciniegas. Design patterns in xml applications. *http://www.xml.com/pub/a/2000/01/19/feature/index.html*.

[2] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 1984.

[3] B. Beizer. *Software Testing Techniques, 2nd Ed.* Van Nostrand Reinhold, 1990.

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

[5] L. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optiomal integration test order. *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 43–55, 2002.

[6] L. Briand, Y. Labiche, and Y. Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. *12th International Symposium on Software Reliability Engineering*, 2001.

[7] L. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategie. *IEEE Transactions on Software Engineering*, 29(7), 2003.

[8] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy. In *Proceedings of 9th European Conference on Object-oriented Programming (ECOOP'95)*, pages 77–101, 1995.

[9] C. Fu, A. Milanova, B. G. Ryder, and D. Wonnacott. Robustness Testing of Java Server Applications. *IEEE Transactions on Software Engineering*, 31(4):292–311, Apr. 2005.

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

[11] C. Hammer and G. Snelting. An improved slicer for java. In *PASTE*, pages 17–22, 2004.

[12] V. L. Hanh, K. Akif, Y. L. Traon, and J.-M. Jzquel. Selecting an efficient object-oriented integration testing strategy: An experimental comparison of actual strategies. *Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001.

[13] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE*, pages 54–61, 2001.

[14] D. Hisley, M. J. Bridges, and L. L. Pollock. Static interprocedural slicing of shared memory parallel programs. In *PDPTA*, pages 658–664, 2002.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedual slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–61, 1990.

[16] http://bddbddb.sourceforge.net/. bddbddb:bdd-based deductive database.

[17] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Symposium on Principles of Database Systems*, pages 299–313, 1990.

[18] 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.

[19] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck, and M.-H. Durand. Testing levels for object-oriented software. *Proceedings of the 22nd international conference on Software engineering (ICSE-22)*, pages 136–145, 2000.

[20] L. Larsen and M. Harrold. Slicing object-oriented software, 1996.

[21] O. Lhotak and L. Hendren. Scaling java points-to analysis using spark. *International Conference on Compiler Construction*, 2003.

[22] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In Proceedings of the International Conference on Software Maintenance, 1998.

[23] D. Liang and M. J. Harrold. Equivalence analysis and its application in improving the efficiency of program slicing. *ACM Trans. Softw. Eng. Methodol.*, 11(3):347–383, 2002.

[24] A. Milanova, A. Rountev, and B. G. Ryder. Constructing precise object relation diagrams. In *ICSM*, pages 586–595, 2002.

[25] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[26] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing, 2nd Edition*. John Wiley and Sons, 2004.

[27] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[28] D. Perry and G. Kaiser. Adequate testing and object- oriented programming. *J. Object-Oriented Programming*, 2(5):13–19, 1990.

[29] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, Sept 1990.

[30] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. In *Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications*, pages 43–55, 2001.

[31] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Proceedings of the Twelveth International Conference on Compiler Construction, pages 126–137, April 2003. invited paper.

[32] M. Sable. Soot: a java optimization framework. See `http://www.sable.mcgill.ca/soot/`.

[33] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[34] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 432–441, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[35] 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, Oct. 2000.

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

[37] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[38] P. Tonella, G. Antonio, R. Fiutem, and E. Merlo. Flow insensitive c++ pointers and polymorphism and its application to slicing. *Proc. 18th IEEE Int'l Conf. Software Eng. (ICSE)*, 1997.

[39] Y. L. Traon, T. Jeron, J.-M. Jezequel, and P. Morel. Efficient object-oriented integration and regression testing. *IEEE Trans. Reliability*, 49(1):12–25, 2000.

[40] UML.org. Omg unified modeling language specification version 1.5, March 2003.

[41] UML.org. Uml 2.0 infrastructure specification version 2.0, final adopted version, Sept 2003.

[42] M. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[43] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *In Proceedings of the 2004 ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2004.

[44] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[45] W. Zhang and B. Ryder. A Practical Algorithm for Interclass Testing Dependence. Technical Report DCS-TR-574, Dept. of Comp. Sci., Rutgers Univ., Apr 2005.