Fragment Class Analysis for Testing of Polymorphism in Java Software

Atanas Rountev
Department of Computer and Information Science
The Ohio State University
rountev@cis.ohio-state.edu

Ana Milanova
Department of Computer Science
Rutgers University
milanova@cs.rutgers.edu

Barbara G. Ryder
Department of Computer Science
Rutgers University
ryder@cs.rutgers.edu

Abstract

Adequate testing of polymorphism in object-oriented software requires coverage of all possible bindings of receiver classes and target methods at call sites. Tools that measure this coverage need to use class analysis *to compute the coverage requirements. However, traditional whole-program class analysis cannot be used when testing partial programs. To solve this problem, we present a general approach for adapting whole-program class analyses to operate on program fragments. Furthermore, since analysis precision is critical for coverage tools, we provide precision measurements for several analyses by determining which of the computed coverage requirements are actually feasible. Our work enables the use of whole-program class analyses for testing of polymorphism in partial programs, and identifies analyses that compute precise coverage requirements and therefore are good candidates for use in coverage tools.*

1 Introduction

Testing of object-oriented software presents new challenges due to features such as inheritance, polymorphism, dynamic binding, and object state [6]. Programs contain complex interactions among sets of collaborating objects from different classes. These interactions are greatly complicated by object-oriented features such as *polymorphism*, which allows the binding of an object reference to objects of different classes. While this is a powerful mechanism for producing compact and extensible code, it creates numerous fault opportunities [6].

Polymorphism is common in object-oriented software—for example, polymorphic bindings are often used instead of `case` statements [15, 7]. However, code that uses polymorphism can be hard to understand and therefore fault-prone—for example, understanding all possible interactions between a message sender and a message receiver under all possible bindings can be challenging for programmers. The sender of a message may fail to meet all preconditions for all possible bindings of the receiver [7]. A subclass in an inheritance hierarchy may violate the contract of its superclasses; clients that send polymorphic messages to this hierarchy may experience inconsistent behavior. For example, an inherited method may be incorrect in the context of the subclass [27], or an overriding method may have preconditions and postconditions different from the ones for the overridden method [6]. In deep inheritance hierarchies, it is easy to forget to override methods for lower-level subclasses [11]; clients of such hierarchies may experience incorrect behavior for some receiver classes. Changes in server classes may cause tested and unchanged client code to fail [7].

1.1 Coverage Criteria for Polymorphism

Various techniques for testing of polymorphic interactions have been proposed in previous work [37, 22, 21, 25, 10, 7, 3]. These approaches require testing that exercises *all possible polymorphic bindings* for certain elements of the tested software. Such requirements can be encoded as coverage criteria for testing of polymorphism. In this paper we focus on two such criteria. The *receiver-classes* criterion (denoted by *RC*) requires exercising of all possible classes of the receiver object at a call site [22, 21, 25, 10]. The *target-methods* criterion (denoted by *TM*) requires exercising of all possible bindings between a call site and the methods that may be invoked by that site [37, 7]. (Both criteria are discussed in more detail in Section 2.)

The testing requirements encoded by the *RC* and *TM* criteria have been advocated by several authors [37, 22, 21, 25, 10, 7].[1] For example, Binder points out that "just as we would not have high confidence in code for which only a small fraction of the statements or branches have been exercised, high confidence is not warranted for a client of a polymorphic server unless all the message bindings generated by the client are exercised" [7]. There is existing evidence that such criteria are better suited for detecting object-oriented faults than the traditional statement and branch coverage criteria [10].

1.2 Class Analysis for Coverage Tools

The use of coverage criteria is impossible without tools that automatically measure the coverage achieved during testing. To compute the *RC* and *TM* coverage requirements, a tool needs to determine the possible classes of the receiver object and the possible target methods for each call site. In the simplest case, this can be done by examining the class hierarchy—i.e., by considering all classes in the subtree rooted at the declared type of the receiver. While not explicitly stated, it appears that all previous work [37, 22, 21, 25, 10, 7] uses this approach to determine the coverage requirements.

Existing work on static analysis for object-oriented languages shows that using the class hierarchy to determine possible receiver classes may be overly conservative—i.e., not all subclasses may be actually possible. Such imprecision has serious consequences for coverage tools. Because of infeasible coverage requirements, high coverage can never be achieved regardless of the testing effort. In this case the coverage metrics become hard to interpret: is the low coverage due to inadequate testing, or is it due to infeasible coverage requirements? This problem seriously compromises the usefulness of the coverage metrics. In addition, the person who creates new test cases may spend significant time and effort trying to determine the appropriate test cases, before realizing that it is impossible to achieve the required coverage. This situation is unacceptable because human time and attention are much more expensive than computing time.

To address these problems, we propose to use *class analysis* to compute the coverage requirements. Class analysis is a static analysis that determines the classes of all objects to which a given reference variable may point. While initially developed in the context of optimizing compilers for object-oriented languages, class analysis also has a variety of applications in software engineering tools. In a coverage tool for testing of polymorphism, class analysis can be used to determine what are the classes of the objects that variable `x` may refer to at call site `x.m()`; from this it is trivial to compute the *RC* and *TM* criteria for the call site. There is a large body of work on various class analyses with different tradeoffs between cost and precision [26, 1, 2, 28, 14, 5, 17, 13, 9, 29, 32, 34, 38, 35, 19, 31, 16, 23]. However, there has been no previous work on using these analyses for the purposes of testing of polymorphism.

1.3 Fragment Class Analysis

The existing body of work on class analysis cannot be used directly to compute the *RC* and *TM* coverage requirements in a coverage tool. The key problem is that the vast majority of existing class analyses are designed as *whole-program analyses*—i.e., analyses that process complete programs. However, testing is rarely done only on complete programs, and many testing activities are performed on partial programs. Any realistic coverage tool should be able to work on partial programs, and therefore cannot incorporate a whole-program class analysis.

To solve this problem, we need to design class analysis that can operate on fragments of programs rather than on complete programs. We refer to such analysis as *fragment class analysis*. The first contribution of this paper is *a general method for constructing fragment class analyses for the purposes of testing of polymorphism in Java*. Using this method, fragment class analyses can be derived from a wide variety of existing (and future) whole-program class analyses [2, 5, 17, 13, 29, 34, 35, 38, 19, 31, 16, 23]. The significance of this technique is that it allows tool designers to adapt available technology for whole-program class analysis to be used in coverage tools for testing of polymorphism in partial programs.

1.4 Absolute Analysis Precision

Analysis precision is a critical issue for the use of class analysis in coverage tools. Less precise analyses compute less precise coverage criteria—i.e., some of the coverage requirements may be impossible to achieve. As discussed earlier, infeasible coverage requirements present a serious problem for coverage tools: the coverage metrics become hard to interpret, and tools users may waste time and effort trying to achieve higher coverage.

To justify the use of a particular class analysis, we need to ensure that few (if any) spurious classes are reported by that analysis. The key problem is that previous work on class analysis only addresses the issue of *relative* analysis precision: how does the solution computed by analysis Y compare to the solution computed by analysis X? However, we need information about *absolute* analysis precision: what part of the analysis solution is infeasi-

[1] Other coverage criteria for polymorphism are also possible. For example, in addition to *RC*, [22] proposes coverage of all possible classes for the senders and the parameters of a message. Our work can be trivially extended to handle such criteria.

ble? The second contribution of this paper is *an empirical evaluation of the absolute precision of four fragment class analyses*. These analyses are based on four well-known whole-program class analyses: Class Hierarchy Analysis (CHA) [12], Rapid Type Analysis (RTA) [5], 0-CFA [33, 16], and Andersen-style points-to analysis [4, 34, 19, 31]. In our experiments we determined manually what parts of the analysis solution were actually infeasible. This information is essential for deciding which analysis to use in a coverage tool; however, to the best of our knowledge, such metrics of absolute precision are not available in any previous work on class analysis.

Our results show that simpler analyses such as CHA and RTA do not provide sufficient precision for the purposes of testing of polymorphism, while more advanced analyses such as 0-CFA and Andersen-style points-to analysis achieve very good precision. These findings lead to two important conclusions. First, our evaluation of CHA and RTA shows that analysis imprecision *can be a serious problem*, and it should be a primary concern when designing coverage tools. Second, our results indicate that more advanced analyses such as 0-CFA and Andersen's analysis *can achieve high absolute precision*, which makes them good candidates for inclusion in coverage tools.

1.5 Contributions

- We present a general approach for constructing fragment class analyses from a wide variety of existing whole-program class analyses. This method enables the designers of coverage tools to use whole-program class analyses for the purposes of testing of polymorphism in partial programs.

- We present an empirical evaluation of the absolute precision of four fragment class analyses. These are the first available results that measure absolute analysis precision, and they provide essential insights for constructing high-quality coverage tools for testing of polymorphism.

Outline. Section 2 describes our coverage tool for testing of polymorphism in Java. Section 3 presents the method for constructing fragment class analyses. The experimental results are described in Section 4. Section 5 discusses related work, and Section 6 presents conclusions and future work.

2 A Coverage Tool for Java

We have built a test coverage tool for Java that supports the *RC* and *TM* coverage criteria. In the context of this tool we have implemented and evaluated several fragment class analyses. In the future we plan to use the tool as the basis for investigations of other problems related to the testing of polymorphism, and more generally, problems related to the testing of object-oriented software.

```
class A { public void m() {..} }
class B extends A { public void m() {..} }
class C extends A {..}
A a;
.....
ci: a.m(); // a may refer to instances of A, B, or C
           // RC(ci) = {A,B,C}  TM(ci) = {A.m,B.m}
```

Figure 1. *RC* and *TM* coverage criteria.

To illustrate the two criteria, consider the Java classes in Figure 1. For the purpose of this example, suppose that reference variable `a` may refer to instances of classes `A`, `B`, or `C`. The *RC* criterion requires testing of call site `a.m()` with each of the three possible classes of the receiver object. Similarly, the *TM* criterion requires testing that invokes each of the two possible target methods (i.e., both `A.m` and the overriding `B.m`). Clearly, *RC* subsumes *TM*.

The input of the tool contains a set Cls of classes that will be tested, as well as a set Int of methods and fields from classes in Cls. These methods and fields are listed by the tool user and they define the interface to the particular functionality that is currently being tested. In general, Int could contain a small subset of all fields and methods from Cls; this corresponds to the case when the user is interested in testing only a specific subset of the functionality provided by the classes from Cls.

A *test suite for* Int is any arbitrary Java class that tests Int (i.e., calls methods from Int and reads/writes fields from Int) and does not access any methods/fields from Cls that are not in Int. We denote by $AllSuites(Int)$ the set of all possible test suites for Int; clearly, this set is infinite. We assume that Cls is closed with respect to Int: for any arbitrary suite $S \in AllSuites(Int)$, any class that could be referenced during the execution of S is included in Cls. In other words, we consider test suites that only test interactions among classes from the given set Cls. In general, classes from Cls could potentially interact with unknown classes from outside of Cls (e.g., with unknown future subclasses of $C \in Cls$). However, at the time the testing is performed, these unknown classes are not available and interactions with them cannot be exercised; therefore, we do not consider test suites whose execution involves such unknown classes.[2]

In addition to Cls and Int, the tool takes as input one particular test suite $T \in AllSuites(Int)$. As output, the tool reports the coverage achieved by T with respect to the *RC* and *TM* criteria.

[2] If the tester has created stub classes to simulate unknown external classes during testing, the stubs should be included in Cls.

```
package station;
public abstract class Link{
   public abstract void
       transmit(String message); }
class NormalLink extends Link { ...  }
class PriorityLink extends Link { ...  }
class SecureLink extends Link { ...  }
class LoggingLink extends Link { ...  }

public class Station {
   private Link link = new NormalLink();
   private int msg_id = 0;
   public void sendMessage(String m) {
  c1: link.transmit(msg_id++ + " " + m);
      if (msg_id==10)
         link = new PriorityLink(); }
   public void report(Link l)
      { c2: l.transmit("id = " + msg_id);} }

public class Factory {
   private boolean secure = false;
   public Link getLink() {
      if (secure) return new SecureLink();
      else        return new NormalLink(); }
   public void makeSecure()
          { secure = true; } }
```

Figure 2. Package `station` with two polymorphic call sites c_1 and c_2.

There are four tool components. The *analysis component* processes the classes in *Cls* and computes the requirements according the *RC* and *TM* criteria—i.e., for each call site c, it produces sets *RC*(c) and *TM*(c). More precisely, the analysis answers the following question: for each call site, what may be the receiver classes and target methods with respect to all possible $S \in \mathit{AllSuites}(\mathit{Int})$? In other words, if it is possible to write some test suite that tests Int and exercises a call site c with some receiver class X or some target method m, the analysis should include X in *RC*(c) and m in *TM*(c). These computed coverage requirements are supplied to the *instrumentation component*, which inserts instrumentation at call sites to record the classes of the receiver objects at runtime (using the reflection mechanism in Java). Instrumentation is only inserted at polymorphic call sites—i.e., sites c for which *RC*(c) is not a singleton set. The instrumented code is supplied to the *test harness* which automatically runs the given test suite T. The results of the execution are processed by the *reporting component*, which determines the actual coverage achieved at call sites.

Example. Consider package `station` in Figure 2. Class `Station` models a station that connects to the rest of the system using a variety of links. Initially, messages are transmitted using a normal-priority link. After certain number of messages have been processed, the station starts using a high-priority link. In addition, the station may be required to report its current state on some link provided from the outside. External code may use class `Factory` to gain access to normal or secure links.

```
package harness;
public abstract class Suite
   { public abstract void run(); }

package stationtest;
import station.*;
public class StationTests
 extends harness.Suite {
  public void run() {
    Station s = new Station();
    Factory f = new Factory();
    Link l;
    for (int i = 0; i < 10; i++) {
       s.sendMessage("message " + i);
       l = f.getLink();
       s.report(l); } } }
```

Figure 3. Simplified test suite.

Suppose that we are interested in testing the functionality that package `station` provides to non-package client code. In this case Int contains methods `Station.sendMessage`, `Station.report`, `Factory.getLink`, `Factory.makeSecure`, and `Link.transmit`, plus the constructors of classes `Station` and `Factory`. Given the package and Int, the tool computes sets *RC*(c_i) and *TM*(c_i) for the call sites in `Station`. For example, using one of the class analyses presented later, the analysis component may produce sets *RC*(c_1) $=$ {`NormalLink,PriorityLink`} and *RC*(c_2) $=$ {`NormalLink,SecureLink`} with the corresponding sets *TM*(c_i). Given this information, the instrumentation component inserts instrumentation at the two call sites. At run time this instrumentation records the classes of the receiver objects using method `Object.getClass`.

Suppose that the tool is used to evaluate the test suite from package `stationtest` shown in Figure 3. The test harness automatically loads and executes the test suite, and then the reporting component provides the coverage results to the tool user. In this particular case, the test suite achieves 50% *RC* coverage for call site c_1 because the site is never executed with receiver class `PriorityLink`. Similarly, the *RC* coverage for c_2 is 50% because receiver class `SecureLink` is not exercised. Note that the suite achieves 100% statement and branch coverage for class `Station`, but this is not enough to achieve the necessary coverage of the polymorphic calls inside the class. To achieve 100% coverage for c_1 and c_2, we need to add at least one more iteration to the loop in `StationTests`, and we also need to introduce a call `f.makeSecure()`.

3 Fragment Class Analysis

As discussed in Section 1.3, whole-program class analyses cannot be used directly in our coverage tool because they cannot be applied to partial programs. In this context, we need fragment class analysis—that is, analysis that can be used to analyze fragments of programs rather than complete programs.

In this section we describe a general method for constructing fragment class analyses for the purposes of testing of polymorphism in Java. The method allows these fragment analyses to be derived from whole-program class analyses. Our approach can be applied to a large number of existing whole-program class analyses [2, 5, 17, 13, 29, 34, 35, 38, 19, 31, 16, 23]. The fragment analyses constructed with this method can be used in coverage tools to compute the requirements of the *RC* and *TM* coverage criteria.

Our approach is designed to be used with existing (and future) whole-program flow-insensitive class analyses. *Flow-insensitive* class analyses do not take into account the flow of control within a method, which makes them less costly than flow-sensitive analyses. The approach is applicable both to context-insensitive and to context-sensitive analyses. *Context-insensitive analyses* do not attempt to distinguish among the different invocation contexts of a method. This category includes Rapid Type Analysis (RTA) by Bacon and Sweeney [5], the XTA/MTA/FTA/CTA family of analyses by Tip and Palsberg [38], Declared Type Analysis and Variable Type Analysis by Sundaresan et al. [35], the p-bounded and p-bounded-linear-edge families of class analyses due to DeFouw et al. [13, 16], 0-CFA [33, 16], 0-1-CFA [17], Steensgaard-style points-to analyses [29, 19], and Andersen-style points-to analyses [34, 19, 31]. Our approach can be applied to all of these context-insensitive whole-program class analyses.

Context-sensitive analyses attempt to distinguish among different invocation contexts of a method. As a result, such analyses are potentially more precise and more expensive than context-insensitive analyses. In *parameter-based* context-sensitive class analyses, calling context is modeled by using some abstraction of the values of the actual parameters at a call site. *Call-chain-based* context-sensitive class analyses represent calling context using a vector of k enclosing call sites. Our approach can be applied both to parameter-based analyses (e.g., the Cartesian Product algorithm due to Agesen [2], the Simple Class Set algorithm by Grove et al. [17], and the parameterized object-sensitive analyses by Milanova et al. [23]) and to call-chain-based analyses (e.g., the standard k-CFA analyses [33, 16], as well as the k-1-CFA analyses by Grove et al. [17, 16]).

3.1 Structure of Fragment Class Analysis

Recall that the input to the tool contains a set of classes Cls, as well as a set Int of methods and fields from Cls that define the interface to the particular functionality that is currently being tested. A test suite for Int is an arbitrary Java class that calls methods from Int, reads/writes fields from Int, and does not access any methods/fields from Cls that are not in Int. $AllSuites(Int)$ is the infinite set of all possible test suites for Int.

The tool needs to compute the coverage requirements according to the *RC* and *TM* criteria—that is, for each method call site, to determine what may be the receiver classes and target methods with respect to all $S \in AllSuites(Int)$. More precisely, if it is possible to write some test suite for Int that exercises a call site c with some receiver class X or some target method m, X should be included in *RC*(c) and m should be included in *TM*(c).

To compute *RC*(c) and *TM*(c), the tool needs to use fragment class analysis. We define an entire family of such analyses in the following manner: first, we create *placeholders* that serve as representatives for the unknown code from all possible test suites $S \in AllSuites(Int)$. During the analysis, the placeholders simulate the potential effects of this unknown code. After creating the appropriate placeholders, the fragment analysis adds them to the tested classes, treats the results as a complete program, and analyzes it using some whole-program class analysis. It is important to note that the created placeholders are *not* designed to be executed as an actual test suite; they are only used for the purposes of the fragment class analysis.

3.2 Placeholders

In our approach we create a placeholder `main` method that contains a variety of placeholder statements, as shown in Figure 4. For each class $X \in Cls$, there is a placeholder variable ph_X that serves as a representative for all unknown external reference variables of type X (i.e., all such variables that may occur in some test suite). Different placeholder statements represent different kinds of statements that could occur in the unknown code from some test suite. For example, `ph_X = new X()` represents the fact that the unknown code may create instances of `X` and assign them to reference variables of type `X`. There are also placeholder statements that represent the effect of accessing fields and methods from Int. Finally, the last two categories of placeholder statements represent the possible effects of assigning variables of one type to variables of another type (including the effects of possible casting). Note that since we are targeting flow-insensitive analyses, the ordering of placeholder statements is irrelevant. For brevity, Figure 4 does not show the placeholder statements for non-default

```
main() {
    // placeholder variable ph_X for every class X ∈ Cls
    X ph_X;
    // for every class X whose constructor is in Int
    ph_X = new X();
    // for every field f ∈ Int declared in class X with type Y
    ph_Y = ph_X.f; ph_X.f = ph_Y;
    // for every method m ∈ Int declared in class X
    // with signature W m(Y,..,Z)
    ph_W = ph_X.m(ph_Y,..,ph_Z);
    // for every subclass Y of class X
    ph_X = ph_Y; ph_Y = (Y)ph_X;
}
```

Figure 4. Placeholder main method and placeholder statements.

constructors, static methods and fields, etc. The actual implementation of fragment class analyses used in our experiments handles the entire Java language.

Example. Consider again package `station` in Figure 2. Suppose that we are interested in testing the functionality that `station` provides to non-package client code. In this case the interface *Int* contains methods `Station.sendMessage`, `Station.report`, `Factory.getLink`, `Factory.makeSecure`, and `Link.transmit` (plus the constructors of `Station` and `Factory`). Given the package and *Int*, the fragment analysis creates the placeholders shown in Figure 5. Placeholder `main` is then added to `station`, and the result can be analyzed using some whole-program class analysis. In Section 3.4 we present examples of the solutions computed by two such whole-program analyses.

3.3 Analysis Correctness

A fragment class analysis is *correct* if and only if the following property holds: if there exists a test suite $S \in AllSuites(Int)$ whose execution exercises a call site c with some receiver class X, the analysis should report that X is a possible receiver class for c. This implies correctness both with respect to the *RC* criterion and the weaker *TM* criterion. We have proven that this property holds for any fragment analysis that is derived from one of the whole-program flow-insensitive analyses listed in the beginning of Section 3 [2, 5, 17, 13, 29, 34, 35, 38, 19, 31, 16, 23]. This result enables the use of a large body of existing work on whole-program class analysis for the purposes of testing of polymorphism.

The proof of this claim is based on a general framework for whole-program class analysis defined by Grove et al. [17, 16]. We have proven the correctness property for two particular precise context-sensitive instantiations of this framework (one parameter-based and one call-chain-based).

```
import station;
main() {
      Station ph_Station;
      Factory ph_Factory;
      Link ph_Link;
      String ph_String;
      ph_Station = new Station();
      ph_Factory = new Factory();
      ph_String = new String();
      ph_Station.sendMessage(ph_String);
      ph_Station.report(ph_Link);
      ph_Link = ph_Factory.getLink();
      ph_Factory.makeSecure();
      ph_Link.transmit(ph_String);
}
```

Figure 5. Placeholders for package `station`.

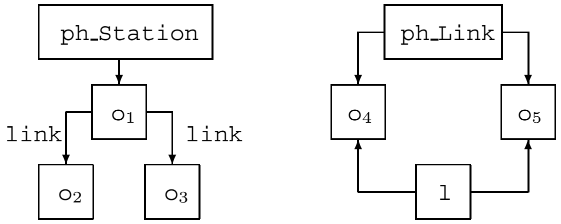
This result implies correctness for *any framework instance* that is less precise than one of these two specific precise instances. In particular, this guarantees the correctness of any fragment analysis that is based on one of the existing whole-program analyses listed above. Furthermore, correctness is also guaranteed with respect to a large class of future whole-program analyses that may be developed by instantiating the framework. For brevity, we omit further discussion of these claims; more details are available in [30].

3.4 Analysis Precision

The approach presented above allows us to construct fragment class analyses from a large number of existing (and future) whole-program class analyses. The quality of the information produced by the fragment analyses depends on the underlying whole-program analysis.

Consider package `station` in Figure 2. If we simply examine the class hierarchy to determine the possible receiver objects at call sites, we would have to conclude that $RC(c_i)$ contains all four subclasses of `Link`, which is too conservative and will result in infeasible testing requirements. In fact, the tool will never report that more than 50% coverage has been achieved for each of the two call sites in `Station`, even if in reality the achieved coverage is 100%.

Now suppose that we add the placeholders from Figure 5 and we run Rapid Type Analysis (RTA) [5]. RTA is a popular whole-program class analysis that performs class analysis and call graph construction in parallel. It maintains a worklist of methods reachable from `main`, and a set of classes instantiated in reachable methods. In the final solution, the set of classes for a variable `v` is the set of all instantiated subclasses of the declared type of `v`. In this example, RTA determines that class `Factory` is instantiated in `main`. This implies that call site `ph_Factory.getLink()` may be executed with an in-



o_1: `new Station()` in `main`

o_2: `new NormalLink()` in `Station`

o_3: `new PriorityLink()` in `Station`

o_4: `new SecureLink()` in `Factory`

o_5: `new NormalLink()` in `Factory`

Figure 6. Some points-to edges computed by Andersen's analysis.

stance of `Factory`, which means that method `getLink` is reachable from `main`. While processing the body of `getLink`, the analysis determines that `NormalLink` and `SecureLink` are instantiated. Similarly, because `Station` is instantiated in `main`, `sendMessage` is determined to be reachable, which implies that `PriorityLink` may also be instantiated. At the end, RTA determines that the only instantiated subclasses of `Link` are `NormalLink`, `PriorityLink`, and `SecureLink`, and therefore $RC(c_i)$ contains only these three classes. Unlike analysis of the class hierarchy, RTA is capable of filtering out the infeasible receiver class `LoggingLink`. Still, some imprecision remains because infeasible class `SecureLink` is reported for c_1 and infeasible class `PriorityLink` is reported for c_2.

As another example, suppose that the fragment analysis uses Andersen's whole-program points-to analysis for Java [34, 19, 31]. This analysis constructs a *points-to graph* in which the nodes represent reference variables and objects, and the edges represent points-to relationships between the nodes. Figure 6 shows some of the edges in the points-to graph computed for our example. Each name o_i represent the run-time objects allocated by a particular `new` expression. (Full description of the analysis and the computed points-to solution is beyond the scope of this paper.) The points-to graph shows that field `link` may only refer to instances of `NormalLink` and `PriorityLink`, and therefore these two classes are included in $RC(c_1)$. Similarly, the points-to graph shows that $RC(c_2)$ contains `NormalLink` and `SecureLink`. Table 1 summarizes the solutions computed by the different analyses. The last row shows which receiver classes are actually feasible—i.e., which classes can be exercised by some test suite $S \in AllSuites(Int)$.

Any class analysis could potentially compute infeasible classes. In this particular case, every receiver class reported by Andersen's analysis is feasible, but in general this need not be true. As discussed in Section 1.4, only analyses that report few (if any) infeasible classes should be used in coverage tools. Otherwise, the coverage metrics become hard to interpret, and tool users may waste time and effort trying to satisfy infeasible testing requirements. Thus, in order to construct high-quality coverage tools for testing of polymorphism, it is necessary to have information about the imprecision of different analyses (i.e., how many infeasible classes they report). However, such measurements of imprecision are not available in previous work on class analysis. One of the major goals of our work was to obtain these measurements for several different class analyses. These results are presented in the next section.

	`Normal`		`Priority`		`Secure`		`Logging`	
	c_1	c_2	c_1	c_2	c_1	c_2	c_1	c_2
Hierarchy	●	●	●	●	●	●	●	●
RTA	●	●	●	●	●	●		
Andersen	●	●	●			●		
Feasible	●	●	●			●		

Table 1. Sets $RC(c_1)$ and $RC(c_2)$ computed by the fragment class analyses.

4 Empirical Results

For our experimental evaluation we used a set of Java packages including the standard packages `java.text` and `java.util.zip`, as well as package `gnu.math` (from `www.gnu.org/software/kawa`) and package `com.lowagie.text` from the `iText` library for creating PDF files (`www.lowagie.com`). We then defined and performed several *testing tasks*. The goal of each task was to write a test suite that exercised some particular functionality provided by these packages. For example, one task exercised the functionality related to identifying boundaries in text (i.e., word boundaries, line boundaries, etc.), as provided by a set of classes from `java.text`. As another example, a task was designed to exercise the functionality from `java.util.zip` related to ZIP files. The first three columns in Table 2 briefly describe the testing tasks and the functionality they exercise.

For each task, we determined the set Int of interface methods and fields for the tested functionality, as well as the set of classes containing the code which implements the functionality. (This was straightforward to do by examining the documentation and the source code). In these classes, we considered all call sites that had more than one possible receiver class according to the class hierarchy. Let $PolySites$ denote the set of all such call sites. Table 2 shows the number of implementing classes for each task and the number of call sites in $PolySites$.

Task	Package	Functionality	#Classes	#PolySites
task1	`java.text`	boundaries in text	12	12
task2	`java.text`	formatting of numbers/dates	13	79
task3	`java.text`	text collation	12	2
task4	`java.util.zip`	ZIP files	8	5
task5	`java.util.zip`	ZIP output streams	8	18
task6	`gnu.math`	complex numbers	8	194
task7	`com.lowagie.text`	paragraphs in PDF docs	24	199
task8	`com.lowagie.text`	lists in PDF docs	24	169

Table 2. Description of testing tasks.

For each task we wrote a test suite that exercised the tested functionality and covered all possible receiver classes for each call site from $PolySites$. Substantial effort was made to ensure that the test suites did in fact achieve *the highest possible coverage*. For each task, two of the authors (working independently of each other) thoroughly examined the code and wrote tests that exercised each possible receiver class. For each call site, the sets of exercised receiver classes obtained by the two people were carefully compared to ensure that there were no differences. As a result of this effort, for each task we had a test suite that exercised all possible receiver classes and target methods for each call site in $PolySites$.

Once we had test suites that achieved the highest possible coverage, we measured the coverage statistics reported by the tool for these suites. These statistics were based on the output of the fragment class analysis used by the tool: the analysis computed a set of possible classes/methods for each $c \in PolySites$, and the tool reported what percentage of these classes/methods was actually exercised by the test suite. In general, this reported coverage may be less than 100% because the analysis produces *RC* and *TM* requirements that are overestimates of the coverage that is actually possible—that is, the analysis may report infeasible receiver classes and infeasible target methods. Clearly, the goal of tool designers should be to use a class analysis that produces few infeasible classes/methods. As a precision metric we used the coverage that was reported by the tool for our test suites; of course, these suites in reality exercised all possible classes and methods. The better the precision of the analysis, the higher the coverage that would be reported by the tool for these suites. Ideally, the analysis would compute only feasible classes and methods, and the tool would report 100% coverage.

4.1 Fragment Class Analyses

For our experiments we evaluated three fragment class analyses. All three analyses were designed using the general approach presented in Section 3: we first created the placeholders from Figure 4, and then we ran the solution engine of a whole-program class analysis.

The first fragment class analysis (denoted by RTA$_f$) was derived from Rapid Type Analysis (RTA) [5]. As discussed in Section 3.4, RTA is a whole-program class analysis that computes an overestimate of the set of classes that are instantiated in methods that are reachable from `main`. This analysis belongs at the lower end of the cost/precision spectrum of class analysis.

The second fragment class analysis (denoted by AND$_f$) was derived from a whole-program points-to analysis for Java [31] which is based on Andersen's points-to analysis for C [4]. This whole-program analysis represents a point at the high end of the cost/precision spectrum for flow- and context-insensitive class analysis. (An example illustrating this analysis is presented in Section 3.4.)

The third fragment class analysis (denoted by 0-CFA$_f$) is derived from a variation of the whole-program points-to analysis from [31]. In this variation, the analysis creates a single object name for all object allocation sites for a given class—i.e., instead of having a separate object name o_i for each `new` expression as in [31], there is a single object name o_C for all expressions "`new C`". This analysis is essentially equivalent to the 0-CFA class analysis [33, 13, 16].[3]

4.2 Analysis Precision

Inside our coverage tool we used the above fragment class analyses to compute the *RC* and *TM* coverage requirements. We then ran our test suites (which in reality exercise all possible classes and methods at polymorphic call sites), and we computed the achieved coverage with respect to *RC-RTA*$_f$, *TM-RTA*$_f$, etc. More precisely, for each analysis, we computed the sum S_1 of the number of possible receiver classes over all sites in $PolySites$ as determined by the analysis, as well as the sum S_2 of the number of actually observed receiver classes at these sites. The tool reported the ratio $C_{RC} = S_2/S_1$ as a coverage metric for the *RC* criterion. A similar ratio C_{TM} was computed for the

[3] The only difference is that our analysis distinguishes among occurrences of the same instance field in different subclasses that inherit that field, while 0-CFA does not make this distinction.

Task	Hierarchy		RTA$_f$		0-CFA$_f$		AND$_f$	
	C_{RC}	C_{TM}	C_{RC}	C_{TM}	C_{RC}	C_{TM}	C_{RC}	C_{TM}
task1	100%	100%	100%	100%	100%	100%	100%	100%
task2	67%	63%	67%	63%	76%	72%	76%	72%
task3	50%	100%	50%	100%	100%	100%	100%	100%
task4	31%	63%	45%	71%	100%	100%	100%	100%
task5	18%	21%	88%	92%	100%	100%	100%	100%
task6	76%	85%	76%	85%	97%	98%	98%	98%
task7	10%	15%	32%	48%	82%	93%	87%	93%
task8	5%	9%	18%	29%	62%	62%	62%	62%

Table 3. Reported coverage. More precise analyses result in higher reported coverage.

TM criterion. The results from these experiments are shown in Table 3. The column labeled "Hierarchy" represents the coverage with respect to the *RC* and *TM* criteria that were computed by just examining the class hierarchy. Class analyses that are more precise result in higher reported coverage percentages. In the best case, the analyses introduce no imprecision (i.e., they do not report infeasible receiver classes), and the reported coverage is 100%.

There are two important conclusions from these results. First, when using the class hierarchy or RTA$_f$ to compute the coverage requirements, there is often a significant number of infeasible receiver classes and target methods. Thus, even for test suites that in reality achieve high coverage, the tool may report low coverage statistics. This situation is clearly unacceptable, and there is a need to use more precise analyses. Second, 0-CFA$_f$ and AND$_f$ perform very well, and in fact in half of the cases they achieve prefect precision. This indicates that these analyses are good candidates for inclusion in realistic coverage tools for testing of polymorphism. To the best of our knowledge, these are the first available empirical results that evaluate the absolute precision of class analysis (i.e., what portion of the analysis solution is infeasible). We believe that such measurements provide essential insights for the designers of coverage tools for testing of polymorphism.

4.3 Analysis Cost

As part of our experiments, we also measured the cost of computing the coverage requirements. All measurements were performed on a 360MHz Sun Ultra-60 machine with 512MB memory. The reported times are the median values out of three runs. Using the class hierarchy or RTA$_f$ had negligible cost (less than 5 seconds for each task). The cost of performing 0-CFA$_f$ and AND$_f$ is shown in Table 4. This cost includes the time to analyze all methods that are directly or transitively reachable from the interface methods, both in classes that implement the tested functionality and in their server classes. The number of these analyzed methods is shown in the last column of Table 4. (The numbers are for AND$_f$; for 0-CFA$_f$, they are essentially the same). Clearly, the two analyses have practical cost, which makes them realistic candidates for use in coverage tools. These results are consistent with similar experiments from [31].

Task	0-CFA$_f$ (sec)	AND$_f$ (sec)	#Methods
task1	4.7	8.6	325
task2	12.8	25.1	752
task3	2.9	5.3	282
task4	5.3	6.4	401
task5	3.6	4.3	280
task6	12.2	35.8	386
task7	13.8	18.1	833
task8	15.4	20.4	810

Table 4. Analysis running times.

5 Related Work

Various authors have recognized the need to test polymorphic relationships by exercising all possible polymorphic bindings [37, 22, 21, 25, 10, 7, 3]. An implicit assumption in this previous work is that the bindings will be determined by examining the class hierarchy—for example, that *RC* coverage of `x.m()` will require covering all subclasses of the declared type of `x`. One key point of our work is that this approach could be overly conservative, and as a result coverage tools may introduce infeasible coverage requirements. As our results indicate, it is essential to use more precise methods for computing the possible bindings. Fortunately, there exists a large body of work on class analysis that can be used to produce more precise coverage requirements. Our work is the first investigation of the use of class analysis for the purposes of testing of polymorphism.

One key problem is that class analyses are typically designed as whole-program analyses, and therefore cannot be used directly for testing of partial programs. Some whole-program class analyses have been adapted to analyze program fragments rather than whole programs. Chatterjee and Ryder [8] present a flow- and context-sensitive points-to analysis for library modules in object-oriented software. The analysis is an adaptation of an earlier whole-program analysis [9]. Sweeney and Tip [36] describe analyses and optimizations for the removal of unused functionality in Java modules. Their work presents a method for performing RTA on program fragments. The approaches from [8] and [36] can be used to compute coverage requirements in tools for testing of polymorphism in partial programs. However, our technique for constructing fragment class analyses (presented in Section 3) is more general and can be applied to a large number of existing whole-program analyses [2, 5, 17, 13, 29, 34, 35, 38, 19, 31, 16, 23]. Furthermore, we present empirical results that evaluate the absolute precision of our fragment analyses and confirm their effectiveness.

Harrold and Rothermel [18] present a method for performing def-use analysis of a given class for the purposes of dataflow-based unit testing in object-oriented languages. Their approach constructs a placeholder driver that represents all possible sequences of method invocations initiated by client code; however, the driver does not take into account the effects of aliasing, polymorphism, and dynamic binding. The placeholder `main` method presented in Section 3 is essentially a placeholder driver that models these features. Thus, in addition to testing of polymorphism, our approach can also be used in tools for dataflow-based testing of individual classes and collections of classes.

We believe that analysis precision is a critical issue for the use of class analysis (or any static analysis) in coverage tools. In previous work, analysis precision is typically evaluated in three ways. One approach is to compare the solutions computed by two or more analyses, in order to determine the relative precision of these analyses—i.e., how analysis X compares with analysis Y. Another approach is to compare the analysis results with the behavior of the program during one particular profile run (e.g., [24, 20]). A third approach is to evaluate the effect of the analysis on a particular client application—for example, the impact on performance due to compiler optimizations. However, in the context of software engineering tools, the key issue is *absolute precision*: how close is the analysis solution to the set of all run-time relationships that are actually possible? Imprecision may lead to significant waste of human time and effort, which ultimately may result in tool rejection. This observation applies not only to coverage tools, but also to other software engineering tools (e.g., for program understanding and verification). Previous work does not contain information about the absolute precision of class analysis, which in our view is a major problem. The precision measurements in Section 4 provide valuable insights for the designers of tools for testing of polymorphism, as well as for other tools that use class analysis.

6 Conclusions and Future Work

In order to construct high-quality coverage tools for testing of polymorphism, it is necessary to use class analysis to compute the coverage requirements. We have developed a general approach that allows tool designers to adapt a wide variety of existing and future whole-program class analyses to be used for testing of partial programs. We also present the first empirical evaluation of the absolute precision of several analyses. Our results lead to two conclusions. First, analysis imprecision can be a serious problem for simpler analyses, and it should be a primary concern for tool designers. Second, more advanced analyses (such as 0-CFA and Andersen's analysis) are capable of achieving high absolute precision, which makes them good candidates for inclusion in coverage tools for testing of polymorphism.

In our future work we would like to evaluate the absolute precision of analyses that are even more precise than 0-CFA and Andersen's analysis. To choose the appropriate analyses, we plan to examine the sources of analysis imprecision. This investigation may suggest the use of existing analyses, or may guide the design of new analysis techniques that target these sources of imprecision. We also plan to obtain additional datapoints for our current analyses, and to evaluate more precise analyses using this extended dataset.

It would be interesting to generalize our approach to flow-sensitive class analyses. We do not anticipate any conceptual difficulties in addressing this problem. Intuitively, it will be necessary to change the structure of our placeholder `main` method to encode all possible sequences of placeholder statements by placing the statements in a switch statement surrounded by a loop.

We also plan to investigate other applications for which fragment class analysis is needed: for example, program understanding and dataflow-based testing of partial Java programs. Such applications require high analysis precision, and it will be necessary to obtain measurements of absolute precision similar to the ones presented in this paper.

Acknowledgments. We would like to thank the ICSE reviewers for their valuable comments, and Matthew Arnold for helping improve an earlier version of the paper. This research was supported by NSF grant CCR-9900988.

References

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

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

[3] R. Alexander and J. Offutt. Criteria for testing polymorphic relationships. In *International Symposium on Software Reliability Engineering*, pages 15–23, 2000.

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

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

[6] R. Binder. Testing object-oriented software: a survey. *Journal of Software Testing, Verification and Reliability*, 6:125–252, Dec. 1996.

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

[8] R. Chatterjee and B. G. Ryder. Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-433, Rutgers University, Apr. 2001.

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

[10] M. H. Chen and M. H. Kao. Testing object-oriented programs—an integrated approach. In *International Symposium on Software Reliability Engineering*, pages 73–83, 1999.

[11] B. Cox. The need for specification and testing languages. *Journal of Object-Oriented Programming*, 1(2):44–47, June 1988.

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

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

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

[15] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

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

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

[18] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. *Symposium on the Foundations of Software Engineering*, pages 154–163, 1994.

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

[20] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the precision of static reference analysis using profiling. In *International Symposium on Software Testing and Analysis*, pages 22–32, 2002.

[21] T. McCabe, L. Dreyer, A. Dunn, and A. Watson. Testing an object-oriented application. *Journal of the Quality Assurance Institute*, 8(4):21–27, Oct. 1994.

[22] R. McDaniel and J. McGregor. Testing the polymorphic interactions between classes. Technical Report 94-103, Clemson University, Mar. 1994.

[23] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[24] M. Mock, M. Das, C. Chambers, and S. Eggers. Dynamic points-to sets. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, 2001.

[25] J. Overbeck. *Integration Testing for Object-Oriented Software*. PhD thesis, Vienna University of Technology, 1994.

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

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

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

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

[30] A. Rountev. *Dataflow Analysis of Software Fragments*. PhD thesis, Rutgers University, Aug. 2002.

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

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

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

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

[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 *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280, 2000.

[36] P. Sweeney and F. Tip. Extracting library-based object-oriented applications. In *Symposium on the Foundations of Software Engineering*, pages 98–107, 2000.

[37] N. N. Thuy. Testability and unit tests in large object-oriented software. In *Proc. 5th International Software Quality Week*. Software Research Institute, 1992.

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