# Fragment Class Analysis for Testing of Polymorphism in Java Software

Atanas Rountev, *Member*, *IEEE Computer Society*, Ana Milanova, *Member*, *IEEE Computer Society*, and Barbara G. Ryder, *Member*, *IEEE Computer Society*

**Abstract**—Testing of polymorphism in object-oriented software may require 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 incomplete 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 for a set of subject components. Our work enables the use of whole-program class analyses for testing of polymorphism in partial programs, and identifies analyses that potentially are good candidates for use in coverage tools.

**Index Terms**—Program analysis, class analysis, test coverage, object-oriented software.

## 1 Introduction

Testing of object-oriented software presents a variety of new challenges due to features such as inheritance, polymorphism, dynamic binding, and object state [1]. 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 [1].

Polymorphism is common in object-oriented software—for example, polymorphic bindings are often used instead of `case` statements [2], [3]. However, code that uses polymorphism can be hard to understand and therefore fault-prone. For example, understanding all possible interactions between a message sender object and a message receiver object under all possible bindings for these objects can be challenging for programmers. The sender object of a message may fail to meet all preconditions for all possible bindings of the receiver object [3]. 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 [4] or an overriding method may have preconditions and postconditions different from the ones for the overridden method [1]. In deep inheritance hierarchies, it is easy to forget to override methods for lower-level subclasses [5]; 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 [3].

### 1.1 Coverage Criteria for Polymorphism

Various techniques for testing of polymorphic interactions have been proposed in previous work [3], [6], [7], [8], [9], [10], [11], [12]. These approaches require testing that exercises *all possible polymorphic bindings* for certain elements of the tested software. 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" [3]. These requirements can be encoded as coverage criteria for testing of polymorphism. There is existing evidence that such criteria are better suited for detecting object-oriented faults than the traditional statement and branch coverage criteria [10].

A *program-based coverage criterion* is a structural test adequacy criterion that defines testing requirements in terms of the coverage of particular elements in the structure of the tested software [13]. Such coverage criteria can be used to evaluate the adequacy of the performed testing and can also provide valuable guidelines for additional testing. In this paper, we focus on two program-based coverage criteria for testing of polymorphism. The *all-receiver-classes* criterion (denoted by *RC*) requires exercising of all possible classes of the receiver object at a call site. The *all-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. Some existing approaches

- A. Rountev is with the Department of Computer Science and Engineering, Ohio State University, 2015 Neil Avenue, Columbus, OH 43210. E-mail: rountev@cis.ohio-state.edu.
- A. Milanova is with the Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180. E-mail: milanova@cs.rpi.edu.
- B.G. Ryder is with the Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854. E-mail: ryder@cs.rutgers.edu.

*Manuscript received 11 Sept. 2003; revised 1 Feb. 2004; accepted 25 Mar. 2004.*
*Recommended for acceptance by L. Dillon.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0143-0903.*

0098-5589/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

explicitly define coverage requirements based on these criteria [3], [6], [8], while, in other approaches, the coverage of receiver classes and/or target methods is part of more general coverage requirements that take into account polymorphism [7], [9], [10], [11], [12]. For example, in addition to *RC*, [7] proposes coverage of all possible classes for the senders and the parameters of a message.

### 1.2 Class Analysis for Coverage Tools

The use of coverage criteria is essentially impossible without tools that automatically measure the coverage achieved during testing. A *coverage tool* analyzes the tested software to determine which elements need to be covered, inserts instrumentation for runtime tracking, executes the test suite, and reports the degree of coverage and the elements that have not been covered. To determine which software elements need to be covered, a coverage tool has to use some form of *source code analysis*. Such an analysis computes the elements for which coverage should be tracked and determines the kind and location of the necessary code instrumentation.

For simple criteria such as statement and branch coverage, the necessary source code analysis is trivial; however, the *RC* and *TM* criteria require more complex analysis. 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 object. It appears that previous work on testing of polymorphism [3], [6], [7], [8], [9], [10], [11], [12] uses this approach (or minor variations of it) to determine the possible receiver classes and target methods at polymorphic calls.

Some of the existing work on static analysis for object-oriented languages (e.g., [14], [15], [16], [17], [18]) shows that using the class hierarchy to determine possible receiver classes may be overly conservative—i.e., not all subclasses may be actually feasible. Such imprecision has serious consequences for coverage tools because the reported 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 the time and attention of a human tester can be very costly compared to computing time.

To address these problems, we propose using *class analysis* to compute the coverage requirements. Class analysis is a static analysis that determines an overestimate of 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 which are the classes of objects that variable `x` may refer to at call site `x.m();` from this information, 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 trade offs between cost and precision [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31]; 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. In contrast, 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, needs analysis techniques beyond traditional whole-program class analyses.

To solve this problem, we need a class analysis that can operate on fragments of programs rather than on complete programs. We refer to such an analysis as a *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 flow-insensitive whole-program class analyses [14], [15], [16], [17], [18], [20], [22], [24], [26], [27], [28], [29], [31]. 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 tool users may waste time and effort trying to achieve higher coverage. 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 $X$? While such comparisons provide useful insights about the relationships between different analyses, they do not address the important question of *absolute* analysis precision: Which parts of an analysis solution are infeasible? The second contribution of this paper is an empirical evaluation of the relative and absolute precision of four fragment class analyses. These analyses are based on four well-known whole-program class analyses: Class Hierarchy Analysis (CHA) [32], Rapid Type Analysis (RTA) [14], 0-CFA [33], [29], and Andersen-style points-to analysis [17], [26], [28], [31], [34]. In our experiments, we determined manually which 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 measurements of absolute precision are not available in any previous work on class analysis.

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

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

Our results indicate that simpler analyses such as CHA and RTA tend to report spurious receiver classes and target methods, while more advanced analyses such as 0-CFA and Andersen-style points-to analysis have the potential to 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 analyses such as 0-CFA and Andersen's analysis have the potential to achieve high absolute precision, which makes them good candidates for further investigation and possible inclusion in coverage tools.

### 1.5 Outline

The rest of the paper is organized as follows: 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 testing of object-oriented software.

To illustrate the two criteria, consider the Java classes in Fig. 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 invocation of each of the two possible target methods (i.e., both `A.m` and the overriding `B.m`). For a polymorphic call site, each possible target method is invoked for at least one possible receiver class; thus, *RC* subsumes *TM*.

### 2.1 Input and Output

The input of the tool contains a set $Cls$ of Java classes and interfaces that will be tested.[1] A subset of $Cls$ is designated as the set of *accessed classes*. Intuitively, an accessed class has methods and fields that may be accessed by future clients of the particular functionality that is currently being tested. If a class is not public [35, Section 6.6], it is accessible only within its declaring package; such a class may still be considered an accessed class if it is possible to have, in this package, some future clients of the tested functionality. For each accessed class $C$, some of its methods and fields (declared in $C$ or inherited from $C$'s superclasses and superinterfaces) are defined as *interface members*. The set of accessed classes and their interface members will be denoted by $Int$; this set defines the interface to the tested functionality. In general, $Int$ could contain a small subset of all classes, fields, and methods from $Cls$, which corresponds to the case when the user is interested in testing only a specific subset of the functionality provided by the classes from $Cls$. Set $Int$ may be obtained in several different ways: A user can manually list its elements; an existing test suite can be analyzed automatically to infer the classes, methods, and fields that constitute the interface to the tested functionality; or the tool can include all nonprivate members for a user-defined "interesting" set of accessed classes. An interface member may potentially have public, protected, or package accessibility.

For the purposes of this paper, a *test suite for* $Int$ is a Java class that contains a set of test cases that exercise $Int$. Without loss of generality, we assume that all test cases are part of a single class whose `main` method serves as a test driver (i.e., it executes all test cases), and this class only references classes from $Cls$ and accesses methods and fields from $Int$. Let $AllSuites(Int)$ denote the set of all such possible test suites for $Int$; clearly, this set is infinite. We assume that $Cls$ is closed with respect to $Int$: For any arbitrary suite $S \in AllSuites(Int)$, all classes, methods, and fields that could be referenced during the execution of $S$ are 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 $Cls$. For example, unknown future subclasses of classes from $Cls$ may override inherited methods and, therefore, instance calls inside $Cls$ may potentially be "redirected" to external code. However, at the time the testing is performed, such 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. If stub classes have been created to simulate unknown external classes during testing, the stubs should be included in $Cls$. In addition to $Cls$ and $Int$, the tool takes as input one particular test suite $T$, and reports the coverage achieved by $T$ with respect to the *RC* and *TM* criteria.

### 2.2 Components

The tool contains four components. The *analysis component* processes the classes in $Cls$ and computes the requirements according the *RC* and *TM* criteria—that is, 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 in $Cls$, what is the set of possible receiver classes and target methods with respect to *all possible* $S \in AllSuites(Int)$? 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 includes $X$ in $RC(c)$ and $m$ in $TM(c)$.

1. Unless stated otherwise, in the rest of the paper, we will use "classes" to refer to both classes and interfaces since, in most cases, the distinction between the two is irrelevant. For brevity, we will also use "methods" to refer to both methods and constructors even though, strictly speaking, constructors are not methods [35].

```
package station;
public abstract class Link
  { public abstract void transmit(String m); }
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; } }
```

Fig. 2. Package station with two polymorphic call sites $c_1$ and $c_2$.

```
package harness;
public abstract class Suite
  { public abstract void run(); }
package stationtest;
import station.*;
public class StationTest 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); } } }
```

Fig. 3. Simplified test suite.

The computed coverage requirements are provided to the *instrumentation component*, which inserts instrumentation at call sites to record the classes of the receiver objects at run time 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 *execution component*, 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.

### 2.3 Example

Consider package station in Fig. 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 a 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.

Suppose we are interested in testing the functionality provided by package station to nonpackage client code. In this case, all public classes (i.e., Link, Station, and Factory) should be considered accessed classes. Set $Int$ contains all public methods in accessed classes: transmit, sendMessage, report, getLink, makeSecure, and the constructors of Station and Factory. (For the purpose of this example, we assume that methods inherited from Object are not relevant.) 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 Andersen's fragment class analysis (presented in Section 3.5), the computed sets are $RC(c_1) = \{\texttt{NormalLink}, \texttt{PriorityLink}\}$ and $RC(c_2) = \{\texttt{NormalLink}, \texttt{SecureLink}\}$ with the corresponding $TM(c_i)$. Given this information, the instrumentation component inserts instrumentation at the two call sites. At runtime, this instrumentation records the receiver classes using Object.getClass.

Suppose that the tool is used to evaluate test suite StationTest shown in Fig. 3. This suite achieves 50 percent *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 percent because receiver class SecureLink is not exercised. Note that the suite achieves 100 percent 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 percent coverage for $c_1$ and $c_2$, we need to add at least one more iteration to the loop in run, 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. The resulting analyses can be used in coverage tools to compute the *RC* and *TM* coverage requirements.

Our approach is designed to be used with existing (and future) whole-program flow-insensitive class analyses. *Flow-insensitive* 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 [14], the XTA/MTA/FTA/CTA family of analyses by Tip and Palsberg [27], Declared Type Analysis and Variable Type Analysis by Sundaresan et al. [16], the $p$-bounded and $p$-bounded-linear-edge families of class analyses due to DeFouw et al. [22], [29], 0-CFA [29], [33], 0-1-CFA [15], Steensgaard-style points-to analyses [24], [28], and Andersen-style points-to analyses [17], [26], [28], [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 with a vector of call sites for the methods that are currently active on the runtime call stack. Our approach can be applied to several parameter-based analyses (the Cartesian Product algorithm due to Agesen [20], the Simple Class Set algorithm by Grove et al. [15], and the parameterized object-sensitive analyses by Milanova et al. [18]) and call-chain-based analyses (the standard $k$-CFA analyses [29], [33], and the $k$-1-CFA analyses by Grove et al. [15], [29]).

### 3.1 Structure of Fragment Class Analysis

Recall from Section 2.1 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 being tested. In addition, $Int$ may contain information about array types that are potentially used in test suites. In general, there is an unbounded number of such array types (e.g., $X[\,]$, $X[\,][\,]$, $X[\,][\,][\,]$, $\ldots$ for some $X$ from $Cls$). To determine which ones are relevant for the tested functionality, we assume that $Int$ contains a list of potentially *instantiated* array types (i.e., types that may occur in `new` expressions) and another list of potentially *accessed* array types (i.e., types that may occur in array access expressions `x[i]`). Knowing that an array type is potentially instantiated is analogous to knowing that some class is potentially instantiated (i.e., $Int$ contains a constructor for the class)—in both cases, this information describes the objects that may be created by test suites. For the purposes of class analysis, knowing that an array type may be accessed by an expression `x[i]` is conceptually similar to knowing that a field in a class is potentially accessed. For example, a statement "`x[i]= y`," where the type of `x` is an accessed array type, is similar to a statement "`x.f = y`" where the type of `x` is an accessed class type. Intuitively, reading or writing an element of an array object is analogous to reading or writing an instance field of a "normal" object.

$AllSuites(Int)$ is the infinite set of possible test suites for $Int$, as defined in Section 2.1. The tool needs to compute the coverage requirements according to the *RC* and *TM* criteria—that is, for each method call site, to determine the set of possible 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 various elements of 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 result 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. Given the information in $Int$ supplied by the tester, the placeholders can be easily constructed automatically by the analysis component of a coverage tool.

There are two categories of placeholders: placeholder *variables* and placeholder *statements*. Both kinds are located inside a placeholder `main` method. Subsequent sections describe the structure of these placeholders and their role in the fragment analysis.

### 3.2 Placeholder Variables

The placeholder variables serve as representatives for unknown external reference variables (i.e., reference variables that may occur in some test suite). A *reference variable* is a variable of reference type. In Java, a reference type is a class type, an interface type, or an array type [35, Section 4.3]. For the purposes of the fragment analysis, an array type with a primitive element type (e.g., `int[]`) is irrelevant. We will use the term *pure reference type* to refer to class types, interface types, and array types whose element types are class/interface types.[2]

The placeholder variables correspond to types that are relevant for possible test suites. We formalize this notion by defining a set $RelevantTypes(Int)$ of pure reference types that are relevant with respect to the tested interface $Int$. For each type $t \in RelevantTypes(Int)$, our approach creates a placeholder variable $ph_t$ that serves as a representative for all unknown external variables of type $t$. The set of relevant types is defined as follows:

2. As defined in the language specification [35, chapter 10], the *component type* of an array type $t$ is the type of the variables contained in the array; this type may itself be an array type. The *element type* of $t$ is obtained by considering the component types until a nonarray type is encountered, e.g., for `int[][]`, the component type is `int[]` and the element type is `int`.

```
import station.*;
main() {
  // Placeholder variables
  Station ph_Station;
  Factory ph_Factory;
  Link ph_Link;
  String ph_String;
  Throwable ph_Throwable;
  // Placeholder statements
  try {
     ph_Station = new Station();
     ph_Factory = new Factory();
     ph_Station.sendMessage(ph_String);
     ph_Station.report(ph_Link);
     ph_Link = ph_Factory.getLink();
     ph_Factory.makeSecure();
     ph_Link.transmit(ph_String);
     ph_String = "abc";
  } catch (Throwable e) {
     ph_Throwable = e;
  }
}
```

Fig. 4. Placeholders for package `station`.

- If the type $t$ of a formal for a method $m \in Int$ is a pure reference type, $t$ is relevant.
- If the return type $t$ for a method $m \in Int$ is a pure reference type, $t$ is relevant.
- If the type $t$ of a field $f \in Int$ is a pure reference type, $t$ is relevant.
- For each accessed class $C$ such that $Int$ contains at least one instance method/field for $C$ (declared in $C$ or inherited from $C$'s superclasses), the class type $C$ is relevant.
- If an instantiated array type $t$ is a pure reference type, $t$ is relevant.
- If an accessed array type $t$ is a pure reference type, $t$ and the component type of $t$ are relevant.
- `java.lang.String` and `java.lang.Throwable` are relevant.

Intuitively, this definition lists the types of all reference variables that may occur in test suites and may affect the flow of reference values in $Cls$—by being, for example, parameters of calls to methods from $Int$. The inclusion of `String` and `Throwable` is necessary for handling of string literals and exceptions, as described shortly. For each relevant type, we create a placeholder variable that represents the effect of variables of that type. All placeholder variables are declared as local variables of the placeholder `main` method.

**Example.** For the definition of $Int$ from Section 2.3, the set of relevant types contains `Station`, `Factory`, `Link`, `String`, and `Throwable`. Fig. 4 shows the declarations of the corresponding placeholder variables.

## 3.3 Placeholder Statements

In addition to the placeholder variables, `main` contains a set of placeholder statements. These statements represent different kinds of statements that could occur in the unknown code from some test suite. Intuitively, the role of the placeholder statements during the class analysis is to "simulate" the possible effects of unknown external code on the flow of reference values. Fig. 4 shows the placeholder statements for the example from Section 2.3. It is important to note that, since we are targeting flow-insensitive class analyses, the ordering of placeholder statements is irrelevant.

### 3.3.1 Method Calls

Consider an accessed class $C$ and one of its (declared or inherited) methods $m \in Int$. There is a single placeholder statement that invokes $m$. If $m$ is an instance method, the placeholder variable for $C$ is used for the receiver expression. The parameters of the call are placeholder variables matching the parameter types of $m$. For example, method `report` in Fig. 2 has a formal parameter of type `Link`, and the corresponding placeholder statement in Fig. 4 uses `ph_Link` as an actual parameter. If a parameter type is not a pure reference type (e.g., `int`), a "dummy" value of that type is used at the call; such values have no effect on the subsequent class analysis.

If the return type of $m$ is a pure reference type, the placeholder statement contains an assignment to the placeholder variable that matches that type. For example, `getLink` in Fig. 2 has return type `Link` and, therefore, the placeholder statement in Fig. 4 assigns the return value of the call to `ph_Link`. In the case when $m$ is a constructor, a `new` expression in introduced, and the result is assigned to the appropriate placeholder variable.

### 3.3.2 Field Accesses

Consider an accessed class $C$ and one of its declared or inherited fields $f \in Int$. There are placeholder statements that read and/or write the field. If $f$ is an instance field with pure reference type $t$, we create a statement "$ph_t = ph_C.f$." In case $f$ is not declared `final`, a statement "$ph_C.f = ph_t$" is also created. If $f$ is a static field of pure reference type $t$ and is declared in class $X$, the two placeholder statements are "$ph_t = X.f$" and "$X.f = ph_t$."

### 3.3.3 Array Creation and Accesses

For each relevant instantiated $n$-dimensional array type $t = X[\,][\,]\ldots[\,]$ with element type $X$, there is a placeholder statement that creates an array of type $t$. The statement has the form "$ph_t = new X[1][\,]\ldots[\,]$." The array creation expression in the statement produces an array with size 1 and with component type either $X$ (if $n = 1$) or the $(n-1)$-dimensional array type $X[\,]\ldots[\,]$ (if $n > 1$). Since the subsequent class analysis does not distinguish among array indices, the array size is irrelevant. This placeholder statement ensures that the class analysis will take into account arrays that may be created in test suites.

For each relevant accessed array type $t$ from $Int$, there are also statements representing accesses of array elements. More precisely, if $w$ is the component type of $t$, `main`

contains statements "$ph_w = ph_t[0]$" and "$ph_t[0] = ph_w$"; the index is irrelevant for the class analysis.

#### 3.3.4 String Literals

There is a placeholder statement that assigns to `ph_String` a string literal (e.g., `"abc"`). This literal represents instances of `String` that correspond to string literals occurring in test suites.

#### 3.3.5 Type Conversions

The Java language defines a set of rules for *compile-time assignment conversions* [35, Section 5.2]. These rules identify pairs $(t_1, t_2)$ of types such that an expression of type $t_1$ can be treated at compile time as if it had type $t_2$ instead. For example, if $Y$ is a subclass of $X$, there is an assignment conversion from the type corresponding to $Y$ to the type corresponding to $X$. Similarly, there is an assignment conversion from $Y$ to each interface that $Y$ implements. Such conversions are implicitly performed at assignment statements and at parameter passing.

To represent the potential effects of these conversions, we consider all pairs of types from $RelevantTypes(Int)$ for which the language defines an assignment conversion. For each such pair $(t_1, t_2)$, `main` contains a placeholder statement of the form "$ph_{t_2} = ph_{t_1}$." For brevity, the complete language rules for these conversions are not shown; a detailed description is available in [35, Section 5.2].

The language also defines a set of rules for *compile-time casting conversions* [35, Section 5.5]. For example, if $Y$ is a subclass of $X$, there is a casting conversion from the type corresponding to $X$ to the type corresponding to $Y$. By definition, all assignment conversions are also valid casting conversions. A casting conversion that is not an assignment conversion requires runtime tests to determine whether the actual reference value is a legitimate value of the new type; if not, a `ClassCastException` is thrown. At compile time, such conversions are achieved through cast expressions. To model the possible effects of these conversions, we consider all pairs of relevant types for which the language defines a casting conversion that is not an assignment conversion. For each such pair $(t_1, t_2)$, we create a placeholder statement "$ph_{t_2} = (t_2)\, ph_{t_1}$" which contains a cast expression. The complete set of language rules for casting conversions is described in [35, Section 5.5].

#### 3.3.6 Exceptions

Since some of the invoked methods may throw checked or unchecked exceptions, all placeholder statements are located inside a statement `try { .. } catch (Throwable e) { ph_Throwable = e;}`. This construct represents the fact that code from test suites may catch exceptions thrown by the tested classes. `Throwable` is the most general type for objects that may be caught by `catch` clauses. Placeholder variable `ph_Throwable` serves as a representative for all reference variables in test suites that may refer to exception objects. Since such variables may be used, for example, as actuals of calls to methods from $Int$, assignment `ph_Throwable = e` enables the potential propagation of caught exceptions.

### 3.4 Analysis Correctness

The previous sections describe our approach for creating a `main` method containing various placeholders. This `main` method is added to the tested classes, and the result is analyzed using some whole-program class analysis. Section 3.5 presents examples of the solutions computed by two such whole-program analyses. In this section, we discuss the correctness of the resulting fragment analysis.

A fragment class analysis is *correct* if and only if the following property holds: If there exists a test suite 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 this property for all fragment analyses derived from the whole-program flow-insensitive analyses listed in the beginning of Section 3 [36], [37]. 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 proposed by Grove et al. [15] and Grove and Chambers [29]. We first define two particular whole-program analyses that are instantiations of this framework. The first analysis, denoted by $\mathcal{A}_p$, is a parameter-based context-sensitive analysis similar to Agesen's Cartesian Product algorithm [20]. The second analysis, denoted by $\mathcal{A}_c$, is a call-chain-based context-sensitive analysis similar to the $k$-1-CFA analysis from [15]. These two analyses are relatively precise instantiations of the framework from [15], [29] and they represent two points at the high end of the precision spectrum for context-sensitive class analysis (with parameter-based sensitivity in $\mathcal{A}_p$ and call-chain-based sensitivity in $\mathcal{A}_c$).

Let $\mathcal{A}'_p$ be the fragment class analysis built on top of $\mathcal{A}_p$. Similarly, let $\mathcal{A}'_c$ be the fragment class analysis that is based on $\mathcal{A}_c$. We have proven the correctness of these two fragment analyses [36], [37]. We believe that the proof techniques employed by the approach may be generalized for other analyses—for example, for fragment side-effect and def-use analyses built on top of existing whole-program analyses.

Consider an arbitrary whole-program class analysis $\mathcal{A}$ that is less precise than $\mathcal{A}_p$ or $\mathcal{A}_c$. Analysis $\mathcal{A}$ always computes a solution that is a superset of the solution computed by $\mathcal{A}_p$ or by $\mathcal{A}_c$. Based on the correctness of $\mathcal{A}'_p$ and $\mathcal{A}'_c$, it is easy to see that the fragment analysis based on $\mathcal{A}$ is also correct. Because of the properties of the framework from [15], [29], each of the whole-program analyses listed in the beginning of Section 3 is either less precise than $\mathcal{A}_p$, or less precise than $\mathcal{A}_c$; this implies the correctness of our approach for all of these analyses. Furthermore, this result applies to any future framework instance that is less precise than $\mathcal{A}_p$ or $\mathcal{A}_c$ and, therefore, correctness is also guaranteed with respect to a large class of future analyses.

### 3.5 Analysis Precision

Consider package `station` in Fig. 2. If we simply examined the class hierarchy to determine the possible receiver objects at call sites, we would have to conclude that $RC(c_i)$ contains

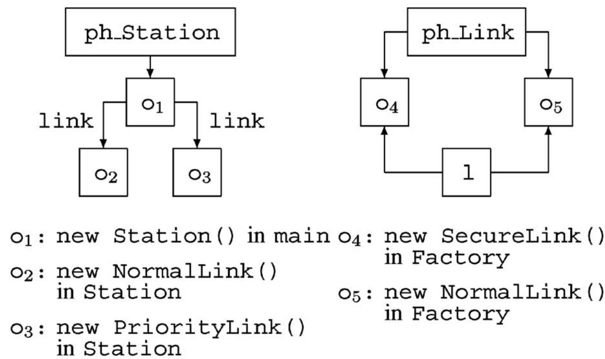


Fig. 5. Some points-to edges computed by Andersen's analysis.

all four subclasses of Link, which is too conservative and will result in infeasible testing requirements. In this case, the tool will never report that more than 50 percent coverage has been achieved for each of the two call sites in Station, even if, in reality, the achieved coverage is 100 percent.

Now, suppose that we add the placeholders from Fig. 4 and we run Rapid Type Analysis (RTA) [14]. RTA is a popular whole-program class analysis that performs class analysis and call graph construction in parallel. It maintains a set 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 instance 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 of this process, 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 [17], [26], [28], [31]. This analysis constructs a *points-to graph* in which nodes represent reference variables and objects, and edges represent points-to relationships between the nodes. Fig. 5 shows some of the edges in the points-to graph computed for our example. Each name $o_i$ represents the runtime objects allocated by a particular new expression. The 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 graph shows that $RC(c_2)$ contains NormalLink and SecureLink.

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 infeasible classes should be used in coverage tools. 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). Unfortunately, measurements of absolute precision are not available in previous work on class analysis. One goal of our work was to obtain such measurements for several different class analyses. These results are presented in the next section.

## 4 Empirical Study

This study focuses on several fragment class analysis techniques derived from popular whole-program class analyses. The purpose of the study is to evaluate these techniques as potential candidates for computing RC and TM coverage requirements in coverage tools. In particular, the key question we want to answer is the following: How precise are the coverage requirements computed by these techniques? In other words, how many infeasible receiver classes and infeasible target methods are included in the computed requirements? Our goal is to evaluate both the relative precision of the techniques with respect to each other, and the absolute precision with respect to a "perfect" baseline. Thus, the manipulated independent variable in our experiments is the class analysis algorithm, and the measured dependent variable is the precision of the coverage requirements.

### 4.1 Analysis Techniques

Our study considers four different choices for the fragment class analysis. Each analysis is derived from a corresponding whole-program analysis, using the approach presented in Section 3. The first analysis, denoted by $\mathrm{CHA}_f$, is based on Class Hierarchy Analysis (CHA) [32]. This approach includes in the coverage requirements each subtype and each overriding method defined in the class hierarchy; therefore, this is the simplest and potentially most imprecise technique. The second fragment class analysis (denoted by $\mathrm{RTA}_f$) is derived from Rapid Type Analysis (RTA) [14]. As discussed in Section 3.5, 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 third fragment analysis, denoted by $0\text{-CFA}_f$, is based on the popular whole-program 0-CFA class analysis [22], [29], [33]. The fourth fragment analysis (denoted by $\mathrm{AND}_f$) is derived from a whole-program points-to analysis for Java [17] which is based on Andersen's points-to analysis for C [34]. (An example illustrating the Andersen-style analysis is presented in Section 3.5.) Both 0-CFA and Andersen-style analysis represent points at the high end of the cost/precision spectrum for flow and context-insensitive class analysis. The difference between the two is that 0-CFA does not attempt to distinguish among different instances of the same class, while Andersen-style analysis

TABLE 1
Description of Testing Tasks

| (1) Task | (2) Package | (3) Functionality | (4) #Classes | | (5) #Methods | | (6) #PolySites |
|---|---|---|---|---|---|---|---|
| | | | (a) CUT | (b) All | (a) CUT | (b) All | |
| $t_1$ | `java.text` | boundaries in text | 12 | 199 | 96 | 2302 | 12 |
| $t_2$ | `java.text` | formatting of numbers/dates | 13 | 205 | 266 | 2504 | 79 |
| $t_3$ | `java.text` | text collation | 12 | 203 | 160 | 2394 | 2 |
| $t_4$ | `java.util.zip` | ZIP files | 8 | 196 | 70 | 2317 | 5 |
| $t_5$ | `java.util.zip` | ZIP output streams | 8 | 194 | 81 | 2284 | 18 |
| $t_6$ | `java.util.zip` | GZIP I/O streams | 6 | 199 | 41 | 2316 | 22 |
| $t_7$ | `gnu.math` | complex numbers | 8 | 205 | 248 | 2624 | 194 |
| $t_8$ | `com.lowagie.text` | paragraphs in PDF docs | 24 | 233 | 345 | 2762 | 199 |
| $t_9$ | `com.lowagie.text` | lists in PDF docs | 24 | 232 | 347 | 2762 | 169 |
| $t_{10}$ | `mindbright.ssh` | SSH client | 60 | 278 | 551 | 3054 | 394 |
| $t_{11}$ | `java.sql` | SQL access | 18 | 206 | 354 | 2592 | 22 |
| $t_{12}$ | `gnu.regexp` | regular expressions engine | 23 | 211 | 144 | 2366 | 71 |
| $t_{13}$ | `com.quiotix.html` | HTML manipulation | 30 | 218 | 299 | 2527 | 60 |
| $t_{14}$ | `jess` | expert system engine | 146 | 502 | 646 | 4724 | 873 |
| $t_{15}$ | `socks` | proxy for SOCKS protocol | 23 | 228 | 232 | 2636 | 241 |
| $t_{16}$ | `jtar.io` | manipulation of tar archives | 21 | 232 | 110 | 2586 | 22 |
| $t_{17}$ | `jflex` | generation of lexical analyzers | 34 | 392 | 316 | 4417 | 533 |
| $t_{18}$ | `gnu.bytecode` | bytecode manipulation | 44 | 246 | 636 | 2982 | 257 |

makes such a distinction in order to improve precision. We used a version of 0-CFA that is a modification of the analysis from [17]. In this modification, 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 [17], there is a single object name $o_C$ for all expressions "`new C`." This analysis is essentially equivalent to the standard 0-CFA class analysis [22], [29], [33]; 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.

## 4.2 Subject Components

For our experimental evaluation, we used a set of publicly available Java packages from a wide range of sources and application domains. We then defined several *testing tasks*. Each task was defined with respect to a particular functionality provided by a package. For example, one task was to exercise the functionality for 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 defined to exercise the functionality from `java.util.zip` related to ZIP files. Columns 1-3 in Table 1 briefly describe the testing tasks and the functionalities they exercise.

For each task, we determined the set $Int$ 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.) This set of classes will be referred to as the *component under test* (CUT) for the corresponding task. Column (4a) in Table 1 shows the number of CUT classes, and (5a) shows the number of methods in these classes. Any class that is directly or transitively referenced by a CUT class could potentially affect the receiver classes and target methods at polymorphic calls inside the CUT. Thus, these classes should also be included in the scope of the class analyses. Column (4b) shows the number of such classes, including the CUT classes. The number of methods in classes from (4b) is shown in (5b).

In all CUT classes, we considered the call sites for which $\mathrm{CHA}_f$ reports more than one possible receiver class. Let $PolySites$ denote the set of all such polymorphic call sites. For each element of this set, our tool computes RC and TM requirements and reports their runtime coverage. The last column in Table 1 shows the size of $PolySites$ for each component.

## 4.3 Measurements of Relative Precision

To measure the precision of the coverage requirements, we defined several metrics. Let $N_{RC}(c, A)$ be the number of possible receiver classes computed by analysis $A$ for call site $c \in PolySites$. Similarly, let $N_{TM}(c, A)$ be the corresponding number of possible target methods. We define

$$N_{RC}(A) = \frac{\sum_{c \in PolySites} N_{RC}(c, A)}{|PolySites|},$$

$$N_{TM}(A) = \frac{\sum_{c \in PolySites} N_{TM}(c, A)}{|PolySites|}$$

as metrics of the size of the coverage requirements computed by analysis $A$, normalized by the number of polymorphic sites in the component. For any pair of analyses $A$ and $A'$, we can define the metric $N_{RC}(A, A') = N_{RC}(A) - N_{RC}(A')$ and the corresponding metric $N_{TM}(A, A')$. These two metrics represent the relative precision of $A$ compared to $A'$. For the 18 components shown in Table 1, we obtained the following metrics:

- $N_{RC}(\mathrm{CHA}_f, \mathrm{AND}_f)$ and $N_{TM}(\mathrm{CHA}_f, \mathrm{AND}_f)$,
- $N_{RC}(\mathrm{RTA}_f, \mathrm{AND}_f)$ and $N_{TM}(\mathrm{RTA}_f, \mathrm{AND}_f)$,
- $N_{RC}(0 - \mathrm{CFA}_f, \mathrm{AND}_f)$ and $N_{TM}(0 - \mathrm{CFA}_f, \mathrm{AND}_f)$.

These metrics compare the theoretically most precise of the four techniques ($\mathrm{AND}_f$) with the remaining three techniques. The analysis of these measurements involved computing some standard descriptive statistics over the set of components: median, arithmetic mean, variation interval, and standard deviation. Since the metrics are based on an absolute scale [38], all of these statistics are meaningful.

Additional analysis of the measurements was performed to allow inferences with some degree of statistical significance. Hypotheses of the form "the probability of $M > x$ is greater than the probability of $M < x$" were formulated and tested statistically for different values of $x$ and for all metrics $M$ listed above. The employed statistical test was the one-tailed paired sign test, with the typical significance level of $\alpha = 0.05$ [39]. The paired sign test is appropriate because it makes no assumptions about the population distribution, and can be applied to small samples.

For a metric $M$, we defined a null hypothesis "the probability of $M > x$ is the same as the probability of $M < x$" and tested it against the alternative hypothesis "the probability of $M > x$ is greater than the probability of $M < x$" for different values of $x$. We then computed the largest value of $x$ for which the null hypothesis can be rejected with significance level $\alpha = 0.05$. Let $\lambda(M)$ denote this largest value. Essentially, $\lambda(M)$ is the greatest lower bound on the median of $M$ that can be inferred at this degree of statistical significance. For example, if $\lambda(N_{RC}(\mathrm{RTA}_f, \mathrm{AND}_f)) = 1.23$, the measurements strongly support the hypothesis that the number of additional receiver classes per polymorphic call site reported by $\mathrm{RTA}_f$ compared to $\mathrm{AND}_f$ will be greater that $1.23$ more often than it will be less than $1.23$.

Hypotheses of the form "the probability of $M < x$ is greater than the probability of $M > x$" were also tested statistically in a similar manner. Let $\lambda'(M)$ denote the smallest value of $x$ for which this hypothesis has statistical significance $\alpha = 0.05$ in the one-tailed paired sign test. This value is the least upper bound on the median of $M$ that can be inferred from the measurements. The interval $(\lambda(M), \lambda'(M))$ characterizes the median value of $M$.

### 4.4 Measurements of Absolute Precision

In order to define a metric for absolute precision, we consider a "perfect" RC/TM solution which contains *all and only* feasible receiver classes and target methods. For such a solution, metrics $N_{RC}(Prec)$ and $N_{TM}(Prec)$ can be defined similarly to the definitions of $N_{RC}(A)$ and $N_{TM}(A)$ presented earlier. For any analysis $A$, $N_{RC}(A, Prec) = N_{RC}(A) - N_{RC}(Prec)$ is a metric of the absolute precision of $A$ with respect to the RC criterion. A similar metric $N_{TM}(A, Prec)$ can be defined as $N_{TM}(A) - N_{TM}(Prec)$.

In general, measurements of absolute precision cannot be obtained automatically through static analysis because any such analysis makes necessarily conservative approximations. To produce such measurements, we performed a set of experiments for tasks $t_1$ through $t_9$ from Table 1. For each task, we wrote a test suite that exercised the tested functionality and covered all feasible 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 feasible 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. The tests were merged into a single test suite that exercised all feasible receiver classes and target methods for each call site in $PolySites$. The runtime coverage achieved by this suite provided a baseline for measuring the absolute precision (i.e., $N_{RC}(A, Prec)$ and $N_{TM}(A, Prec)$) of the four fragment analyses used in the experiments. Similarly to the analysis of relative precision, additional statistical analysis was performed on the metrics of absolute precision. Using the approach described earlier, an interval $(\lambda(M), \lambda'(M))$ was computed for each absolute precision metric $M$. The endpoints of this interval are the greatest lower bound and the least upper bound on the median of $M$ that can be inferred from the measurements for tasks $t_1$ through $t_9$.

The kind of experiment described above is somewhat unusual for program analysis research for two reasons. First, a threat to the validity of the results is the possibility of human error in determining which parts of the analysis solution are feasible. Even though this factor is partially controlled by having two experimenters working independently, in general it is not possible to completely eliminate this threat. Second, the experiments are labor-intensive and thus hard to scale to a large number of subjects, or to subjects with heavy use of polymorphism. This substantial amount of effort was the reason to obtain absolute precision measurements only for half of the tasks in Table 1. Despite these drawbacks, we believe that such experiments provide essential insights for analysis designers and tool builders. Section 4.7 discusses this issue in more detail.

### 4.5 Threats to Validity

As with any empirical study, there are various threats to the validity of our study including threats to conclusion validity, internal validity, external validity, and construct validity [40]. Threats to conclusion validity and internal validity are factors that may invalidate the conclusions about the relationship between independent and dependent variables for the experiment subjects. In our study, the implementation of the analyses is an experiment artifact that may introduce such threats: If the implementation is incorrect, the results of the study may be partially invalid. The probability of this threat is reduced by the extensive

TABLE 2
Coverage Requirements

| Task | $N_{RC}(A)$ | | | | | $N_{TM}(A)$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | CHA$_f$ | RTA$_f$ | 0-CFA$_f$ | AND$_f$ | *Prec* | CHA$_f$ | RTA$_f$ | 0-CFA$_f$ | AND$_f$ | *Prec* |
| $t_1$ | 4.00 | 4.00 | 4.00 | 4.00 | 4.00 | 3.50 | 3.50 | 3.50 | 3.50 | 3.50 |
| $t_2$ | 2.86 | 2.86 | 2.53 | 2.53 | 1.92 | 2.48 | 2.48 | 2.18 | 2.18 | 1.57 |
| $t_3$ | 2.00 | 2.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| $t_4$ | 3.20 | 2.20 | 1.00 | 1.00 | 1.00 | 1.60 | 1.40 | 1.00 | 1.00 | 1.00 |
| $t_5$ | 7.00 | 1.44 | 1.28 | 1.28 | 1.28 | 6.17 | 1.38 | 1.28 | 1.28 | 1.28 |
| $t_6$ | 14.4 | 4.32 | 2.50 | 2.50 | 2.36 | 10.91 | 3.64 | 2.50 | 2.50 | 2.36 |
| $t_7$ | 3.76 | 3.76 | 2.92 | 2.91 | 2.85 | 2.41 | 2.41 | 2.10 | 2.09 | 2.05 |
| $t_8$ | 6.17 | 1.91 | 0.75 | 0.71 | 0.62 | 3.55 | 1.07 | 0.56 | 0.56 | 0.52 |
| $t_9$ | 6.42 | 1.77 | 0.51 | 0.51 | 0.31 | 3.60 | 1.08 | 0.51 | 0.51 | 0.31 |
| $t_{10}$ | 14.32 | 6.60 | 1.82 | 1.82 | — | 1.82 | 1.64 | 1.43 | 1.43 | — |
| $t_{11}$ | 2.77 | 1.55 | 1.32 | 1.32 | — | 1.00 | 1.00 | 1.00 | 1.00 | — |
| $t_{12}$ | 5.73 | 4.73 | 4.10 | 4.10 | — | 3.58 | 3.58 | 3.28 | 3.28 | — |
| $t_{13}$ | 6.07 | 3.95 | 1.98 | 1.98 | — | 2.52 | 2.25 | 1.73 | 1.73 | — |
| $t_{14}$ | 5.53 | 4.41 | 2.94 | 2.94 | — | 2.04 | 1.89 | 1.63 | 1.63 | — |
| $t_{15}$ | 7.20 | 3.48 | 1.69 | 1.69 | — | 3.24 | 2.25 | 1.44 | 1.44 | — |
| $t_{16}$ | 4.00 | 2.36 | 1.86 | 1.86 | — | 2.55 | 1.86 | 1.73 | 1.73 | — |
| $t_{17}$ | 10.02 | 4.22 | 1.16 | 1.16 | — | 2.80 | 1.87 | 1.03 | 1.03 | — |
| $t_{18}$ | 14.4 | 7.37 | 4.20 | 4.20 | — | 2.67 | 2.16 | 1.86 | 1.86 | — |

testing of the implementation in the context of this project and several other projects over the last few years. Another threat is possible human error in deciding which receiver classes are feasible in order to obtain measurements of absolute precision. By having two experimenters obtain these measurements independently, we partially control for this factor. The employed statistical test (the paired sign test) is another potential validity threat. This test is relatively weak and, therefore, has limited ability to reveal patterns in the data. In particular, the lower bounds $\lambda$ and the upper bounds $\lambda'$ may be too conservative—that is, they may provide imprecise characterization of the corresponding metrics. More powerful tests such as the paired t-test and the Wilcoxon test [40] require certain assumptions to be true (e.g., the t-test assumes normal distribution). At present, there is no existing evidence to support such assumptions.

Threats to external validity affect the ability to generalize the results of an experiment. In our study, the source of such threats is the set of subject components. Ideally, the sample of the population should be representative of the entire population to which we want to generalize. In particular, for any factor that may affect the dependent variables (i.e., the precision metrics), the subjects should provide a representative sample with respect to this factor. Examples of such factors are the programming style (in particular, the use of polymorphism) and the application domain. Another potential factor is the size of the subject, even though some anecdotal evidence (e.g., [17], [18]) suggests that there is little correlation between subject size and relative precision. The factors that affect analysis precision have not been identified or quantified in a systematic manner by existing work on program analysis. Therefore, at present, it is impossible to argue that the subjects used in this study (or in any similar study in previous work) constitute a representative sample. To address this threat, we used subjects produced by different developers, presumably using a variety of programming styles, and from different application domains. For the measurements of relative precision, the size of the sample ($n = 18$) is larger and, therefore, somewhat stronger conclusions can be drawn from these results. The measurements of absolute precision use a smaller sample ($n = 9$) and present a weaker basis for generalizations.

Threats to construct validity concern the generalization of the experiment to the concept behind the experiment. In our study, the precision metrics may not necessarily account for all aspects of the costs and benefits faced by tool users. For example, $N_{RC}(A_1, Prec) = 2 \times N_{RC}(A_2, Prec)$ does not imply that a tool user will take twice as long to identify the infeasible RC requirements produced by analysis $A_1$, compared to identifying the infeasible RC requirements produced by analysis $A_2$. Ultimately, experiments with human subjects will be necessary to gain better understanding of the effects of analysis precision on tester productivity.

### 4.6 Results and Interpretation

Table 2 shows the metrics for coverage requirement size, as computed by the different analyses. For tasks $t_1$ through $t_9$, columns $Prec$ also show the metrics for the feasible

TABLE 3
Maximum Reported Coverage

| Task | $N_{RC}(Prec)/N_{RC}(A)$ | | | | $N_{TM}(Prec)/N_{TM}(A)$ | | | |
|---|---|---|---|---|---|---|---|---|
| | CHA$_f$ | RTA$_f$ | 0-CFA$_f$ | AND$_f$ | CHA$_f$ | RTA$_f$ | 0-CFA$_f$ | AND$_f$ |
| $t_1$ | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| $t_2$ | 67% | 67% | 76% | 76% | 63% | 63% | 72% | 72% |
| $t_3$ | 50% | 50% | 100% | 100% | 100% | 100% | 100% | 100% |
| $t_4$ | 31% | 45% | 100% | 100% | 63% | 71% | 100% | 100% |
| $t_5$ | 18% | 88% | 100% | 100% | 21% | 92% | 100% | 100% |
| $t_6$ | 17% | 58% | 95% | 95% | 23% | 69% | 95% | 95% |
| $t_7$ | 76% | 76% | 97% | 98% | 85% | 85% | 98% | 98% |
| $t_8$ | 10% | 32% | 82% | 87% | 15% | 48% | 93% | 93% |
| $t_9$ | 5% | 18% | 62% | 62% | 9% | 29% | 62% | 62% |
| $t_{10}$ | $\leq$ 13% | $\leq$ 28% | — | — | $\leq$ 79% | $\leq$ 87% | — | — |
| $t_{11}$ | $\leq$ 48% | $\leq$ 85% | — | — | — | — | — | — |
| $t_{12}$ | $\leq$ 72% | $\leq$ 87% | — | — | $\leq$ 92% | $\leq$ 92% | — | — |
| $t_{13}$ | $\leq$ 33% | $\leq$ 50% | — | — | $\leq$ 69% | $\leq$ 77% | — | — |
| $t_{14}$ | $\leq$ 53% | $\leq$ 67% | — | — | $\leq$ 80% | $\leq$ 86% | — | — |
| $t_{15}$ | $\leq$ 23% | $\leq$ 49% | — | — | $\leq$ 44% | $\leq$ 64% | — | — |
| $t_{16}$ | $\leq$ 47% | $\leq$ 79% | — | — | $\leq$ 68% | $\leq$ 93% | — | — |
| $t_{17}$ | $\leq$ 12% | $\leq$ 27% | — | — | $\leq$ 37% | $\leq$ 55% | — | — |
| $t_{18}$ | $\leq$ 29% | $\leq$ 57% | — | — | $\leq$ 70% | $\leq$ 86% | — | — |

TABLE 4
Precision Metrics: Descriptive Statistics and Median Bounds

| Metric | Median | Mean | Min | Max | StdDev | Lower bound $\lambda$ | Upper bound $\lambda'$ |
|---|---|---|---|---|---|---|---|
| $N_{RC}$(CHA$_f$,AND$_f$) | 3.34 | 4.58 | 0 | 12.5 | 3.99 | 1.63 | 5.73 |
| $N_{TM}$(CHA$_f$,AND$_f$) | 0.7 | 1.54 | 0 | 8.41 | 2.17 | 0.32 | 1.17 |
| $N_{RC}$(RTA$_f$,AND$_f$) | 1.2 | 1.41 | 0 | 4.78 | 1.24 | 0.63 | 1.8 |
| $N_{TM}$(RTA$_f$,AND$_f$) | 0.3 | 0.37 | 0 | 1.14 | 0.31 | 0.2 | 0.52 |
| $N_{RC}$(0-CFA$_f$,AND$_f$) | 0 | 0.003 | 0 | 0.04 | 0.01 | 0 | 0.01 |
| $N_{TM}$(0-CFA$_f$,AND$_f$) | 0 | 0.0006 | 0 | 0.01 | 0.002 | 0 | 0.01 |
| $N_{RC}$(CHA$_f$, *Prec*) | 2.2 | 3.83 | 0 | 12.05 | 3.9 | 0.9 | 6.11 |
| $N_{TM}$(CHA$_f$, *Prec*) | 0.91 | 2.4 | 0 | 8.55 | 2.88 | 0 | 4.89 |
| $N_{RC}$(RTA$_f$, *Prec*) | 1 | 0.99 | 0 | 1.96 | 0.61 | 0.15 | 1.46 |
| $N_{TM}$(RTA$_f$, *Prec*) | 0.4 | 0.49 | 0 | 1.28 | 0.44 | 0 | 0.91 |
| $N_{RC}$(0-CFA$_f$, *Prec*) | 0.07 | 0.13 | 0 | 0.61 | 0.2 | 0 | 0.2 |
| $N_{TM}$(0-CFA$_f$, *Prec*) | 0.04 | 0.12 | 0 | 0.61 | 0.2 | 0 | 0.2 |
| $N_{RC}$(AND$_f$, *Prec*) | 0.06 | 0.12 | 0 | 0.61 | 0.2 | 0 | 0.2 |
| $N_{TM}$(AND$_f$, *Prec*) | 0.04 | 0.12 | 0 | 0.61 | 0.2 | 0 | 0.2 |

coverage requirements. Table 3 shows the maximum possible coverage that may be reported by the tool if it were to use analysis $A$ to compute the coverage requirements. For example, for task $t_5$, $N_{RC}(\text{CHA}_f) = 7$, but the best possible RC coverage that may be achieved is $N_{RC}(Prec) = 1.28$, which is 18 percent of $N_{RC}(\text{CHA}_f)$; this means that 82 percent of the receiver classes reported by CHA$_f$ are infeasible. For tasks $t_{10}$ through $t_{18}$, upper bounds on the maximum reported coverage can be obtained by using $N_{RC}(\text{AND}_f)/N_{RC}(A)$ and $N_{TM}(\text{AND}_f)/N_{TM}(A)$. The bottom half of Table 3 shows these upper bounds; if only the trivial bound $\leq 100$ percent could be inferred, it is not shown in the table. The top part of Table 4 shows the standard descriptive statistics and the median bounds for

TABLE 5
Analysis Running Times

| Task | 0-CFA$_f$ (sec) | AND$_f$ (sec) | #Methods | Task | 0-CFA$_f$ (sec) | AND$_f$ (sec) | #Methods |
|---|---|---|---|---|---|---|---|
| $t_1$ | 2.0 | 4.0 | 325 | $t_{10}$ | 11.1 | 21.2 | 1161 |
| $t_2$ | 5.0 | 11.5 | 752 | $t_{11}$ | 4.9 | 8.9 | 702 |
| $t_3$ | 1.5 | 2.5 | 282 | $t_{12}$ | 1.9 | 2.6 | 348 |
| $t_4$ | 2.3 | 2.8 | 401 | $t_{13}$ | 3.6 | 4.7 | 652 |
| $t_5$ | 1.9 | 2.1 | 280 | $t_{14}$ | 21.6 | 51.3 | 1638 |
| $t_6$ | 1.3 | 1.4 | 286 | $t_{15}$ | 4.0 | 10.2 | 748 |
| $t_7$ | 5.1 | 13.6 | 386 | $t_{16}$ | 2.9 | 3.2 | 486 |
| $t_8$ | 5.6 | 8.1 | 833 | $t_{17}$ | 14.8 | 52.9 | 1055 |
| $t_9$ | 5.9 | 8.5 | 810 | $t_{18}$ | 10.5 | 18.6 | 1336 |

the relative precision metrics for all tasks. The bottom part of the table contains the same information for the absolute precision metrics for $t_1$ through $t_9$.

The results presented in Tables 2, 3, and 4 should be interpreted in the context of the validity threats discussed in Section 4.5. In particular, the external validity of the study—that is, how the results can be generalized to other subjects—cannot be easily estimated. This problem is common for essentially all program analysis research, where the sample size is typically small (usually $n < 20$), and the subject properties that affect analysis precision are rarely identified and quantified. Our study should be considered as a step in a long-term process of gathering data to support conclusions with some degree of statistical significance.

For the evaluation of relative precision, the results indicate that $\mathrm{CHA}_f$ has the tendency to report spurious receiver classes. For $N_{RC}(\mathrm{CHA}_f, \mathrm{AND}_f)$, the median value is $3.34$ over the 18 tasks, and the lower bound on the median is $1.63$. Since $N_{RC}(\mathrm{CHA}_f, Prec) \geq N_{RC}(\mathrm{CHA}_f, \mathrm{AND}_f)$, this data strongly suggests that $\mathrm{CHA}_f$ should not be used to compute the RC requirements in coverage tools. The results for $N_{RC}(\mathrm{RTA}_f, \mathrm{AND}_f)$, with a median of $1.2$ and a lower bound of $0.63$, indicate that RTA may also be a poor candidate for computing RC requirements. To a lesser degree, the results for TM coverage suggest similar conclusions. Somewhat surprisingly, the measurements for $0\text{-CFA}_f$ strongly indicate that negligible precision improvement should be expected if using $\mathrm{AND}_f$ instead of $0\text{-CFA}_f$.

The evaluation of absolute precision is performed on a smaller sample and, therefore, conclusions based on these results are weaker than the conclusions based on the relative precision metrics. For $N_{RC}(0\text{-CFA}_f, Prec)$, the median value is $0.07$ and the maximum value is $0.61$. The median values are slightly smaller for TM coverage and for $\mathrm{AND}_f$. For four of the nine tasks, both $0\text{-CFA}_f$ and $\mathrm{AND}_f$ achieve perfect precision. These results indicate that these two analyses may be good candidates for future investigation and for potential inclusion in coverage tools for testing of polymorphism.

For completeness, we also measured the cost of computing the coverage requirements. All measurements were performed on a 900MHz Sun Fire-280R machine with 3GB memory. The reported times are the median values out of three runs. Using $\mathrm{CHA}_f$ and $\mathrm{RTA}_f$ has negligible cost (less than 2 seconds). The cost of performing $0\text{-CFA}_f$ and $\mathrm{AND}_f$ is shown in Table 5. 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 (i.e., in classes that are used by the code that implements the functionality). The number of these analyzed methods for $\mathrm{AND}_f$ is shown in the last column of Table 5; for $0\text{-CFA}_f$, the number of analyzed methods is almost the same.

These results should not be interpreted as cost comparison between $0\text{-CFA}_f$ and $\mathrm{AND}_f$ because the differences may be due to properties of our particular implementations. Rather, the results provide an upper bound on the cost of these analyses for the subject components. The primary factor affecting analysis cost is the implementation of the underlying Andersen-style whole-program analysis. Recent work [31] presents efficient techniques for implementing this analysis, with running times in the order of a minute per ten thousand analyzed methods.

### 4.7 Discussion and Conclusions

The goal of our study is to gain insights about the precision of several fragment class analyses. This is important not only for coverage tools, but also for tools for understanding and transformation of object-oriented software. Various approaches for precision evaluation can be employed to provide such insights. Relative precision comparisons can identify analyses that are consistently imprecise and, therefore, may be poor choices for software tools. For example, the results presented earlier indicate that $\mathrm{CHA}_f$ and $\mathrm{RTA}_f$ may be such poor choices. Thus, relative precision comparisons can provide valuable information for tool designers.

The disadvantage of relative precision evaluations is that they can identify analyses that have a high degree of imprecision, but not analyses that have a low degree of imprecision. For example, even though $0\text{-CFA}_f$ and $\mathrm{AND}_f$ are more precise than $\mathrm{CHA}_f$ and $\mathrm{RTA}_f$, this information by itself does not indicate how far away they are from the "perfect" solution. Eventually, this question must be addressed by some form of absolute precision evaluation. One possible approach is to perform studies similar to the

one presented in this work. To the best of our knowledge, these are the first available results that evaluate the absolute precision of class analysis. The study indicates that $0\text{-CFA}_f$ and $\text{AND}_f$ are promising candidates for future investigation. Clearly, long-term gathering of data by us and by other researchers is necessary to obtain conclusive results about the absolute precision of these and other class analyses. We consider our study to be a first step in these investigations.

The absolute precision evaluation in this work is performed through a manual "brute-force" approach. Over a longer period of time and with the participation of more researchers, this can yield a significant body of data. However, it is also necessary to attempt to reduce the cost of this process. One possibility is to define approaches that provide *estimates* of absolute precision and to evaluate the possible error in these estimates using studies similar to ours.[3] Such techniques may require investigation of the sources of class analysis imprecision; for example, certain commonly used object-oriented idioms are a potential source of imprecision [18]. It may be necessary to define metrics that quantify these sources, to build models of their impact on the analysis solution, and to evaluate these models empirically.

## 5 Related Work

Various authors have recognized the need to test polymorphic relationships by exercising all possible polymorphic bindings [3], [6], [7], [8], [9], [10], [11], [12]. The coverage of receiver classes and/or target methods is either needed as an explicit testing goal or as part of more general coverage criteria—for example, criteria based on object-level def-use coverage that takes into account polymorphism [10], [11], [12]. An implicit assumption in this previous work appears to be that the bindings will be determined by examining the class hierarchy—for example, that the possible receiver classes at `x.m()` are the 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. 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 analyses more complicated than CHA 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 [41] 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 [23]. Tip et al. [42] and Sweeney and Tip [43] describe analyses and optimizations for the removal of unused functionality in Java modules. Their work presents a method for performing RTA [14] and XTA [27] on program fragments. Although the approaches from [41] and [42] can be used to compute coverage requirements in tools for testing of polymorphism in partial programs, 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 [14], [15], [16], [17], [18], [20], [22], [24], [26], [27], [28], [29], [31].

Harrold and Rothermel [44] 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 could potentially be useful in tools for dataflow-based testing of individual classes and collections of classes.

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 a particular set of test runs (e.g., [45], [46]). 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, another important issue is absolute precision: How close is the analysis solution to the set of all runtime relationships that are actually possible? Imprecision may lead to a waste of the tester's 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 serious problem. The study from Section 4 is a step toward investigating this issue and gaining insights needed by designers of tools that employ 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 an important 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 more investigation and potentially for subsequent inclusion in coverage tools.

In our future work, we will evaluate the absolute precision of analyses that are even more precise than

3. We would like to thank one of the reviewers for suggesting this approach.

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 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 data set. Additional studies by us and by other investigators will be necessary to obtain conclusive results about the absolute precision of different class analyses. Furthermore, it is important to develop techniques for reducing the cost of absolute precision evaluations, and to consider approaches for obtaining absolute precision estimates.

It would be interesting to generalize our approach to flow-sensitive class analyses. 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 analysis is needed (e.g., program understanding) and to consider other categories of analyses such as side effect analysis and def-use analysis using the theoretical techniques presented in [36], [37].

## Acknowledgments

The authors would like to thank the reviewers for their very thorough comments and suggestions. Their feedback was essential for improving the contents and the presentation of this paper. This research was supported in part by US National Science Foundation grant CCR-9900988.

## References

[1] R. Binder, "Testing Object-Oriented Software: A Survey," *J. Software Testing, Verification and Reliability,* vol. 6, pp. 125-252, Dec. 1996.
[2] M. Fowler, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
[3] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.
[4] D. Perry and G. Kaiser, "Adequate Testing and Object-Oriented Programming," *J. Object-Oriented Programming,* vol. 2, no. 5, pp. 13-19, Jan. 1990.
[5] B. Cox, "The Need for Specification and Testing Languages," *J. Object-Oriented Programming* vol. 1, no. 2, pp. 44-47, June 1988.
[6] N.N. Thuy, "Testability and Unit Tests in Large Object-Oriented Software," *Proc. Fifth Int'l Software Quality Week,* 1992.
[7] R. McDaniel and J. McGregor, "Testing the Polymorphic Interactions between Classes," Technical Report 94-103, Clemson Univ., Mar. 1994.
[8] T. McCabe, L. Dreyer, A. Dunn, and A. Watson, "Testing an Object-Oriented Application," *J. Quality Assurance Inst.,* vol. 8, no. 4, pp. 21-27, Oct. 1994.
[9] J. Overbeck, "Integration Testing for Object-Oriented Software," PhD dissertation, Vienna Univ. of Technology, 1994.
[10] M.H. Chen and M.H. Kao, "Testing Object-Oriented Programs—An Integrated Approach," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 73-83, 1999.
[11] R. Alexander and J. Offutt, "Criteria for Testing Polymorphic Relationships," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 15-23, 2000.
[12] R. Alexander, "Testing the Polymorphic Relationships of Object-Oriented Programs," PhD dissertation, George Mason Univ., 2001.
[13] H. Zhu, P. Hall, and J. May, "Software Unit Testing Coverage and Adequacy," *ACM Computing Surveys,* vol. 29, no. 4, pp. 366-427, Dec. 1997.
[14] D. Bacon and P. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 324-341, 1996.
[15] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call Graph Construction In Object-Oriented Languages," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 108-124, 1997.
[16] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical Virtual Method Call Resolution for Java," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 264-280, 2000.
[17] A. Rountev, A. Milanova, and B.G. Ryder, "Points-To Analysis for Java Using Annotated Constraints," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 43-55, Oct. 2001.
[18] A. Milanova, A. Rountev, and B.G. Ryder, "Parameterized Object Sensitivity for Points-To and Side-Effect Analyses for Java," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 1-11, 2002.
[19] J. Palsberg and M. Schwartzbach, "Object-Oriented Type Inference," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 146-161, 1991.
[20] O. Agesen, "The Cartesian Product Algorithm," *Proc. European Conf. Object-Oriented Programming,* pp. 2-26, 1995.
[21] J. Plevyak and A. Chien, "Precise Concrete Type Inference for Object-Oriented Languages," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 324-340, 1994.
[22] G. DeFouw, D. Grove, and C. Chambers, "Fast Interprocedural Class Analysis," *Proc. Symp. Principles of Programming Languages,* pp. 222-236, 1998.
[23] R. Chatterjee, B.G. Ryder, and W. Landi, "Relevant Context Inference," *Proc. Symp. Principles of Programming Languages,* pp. 133-146, 1999.
[24] C. Razafimahefa, "A Study of Side-Effect Analyses for Java," master's thesis, McGill Univ., Dec. 1999.
[25] E. Ruf, "Effective Synchronization Removal for Java," *Proc. Conf. Programming Language Design and Implementation,* pp. 208-218, 2000.
[26] M. Streckenbach and G. Snelting, "Points-To for Java: A General Framework and an Empirical Comparison," technical report, Univ. Passau, Sept. 2000.
[27] F. Tip and J. Palsberg, "Scalable Propagation-Based Call Graph Construction Algorithms," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 281-293, 2000.
[28] D. Liang, M. Pennings, and M.J. Harrold, "Extending and Evaluating Flow-Insensitive and Context-Insensitive Points-To Analyses for Java," *Proc. Workshop Program Analysis for Software Tools and Eng.,* pp. 73-79, 2001.
[29] D. Grove and C. Chambers, "A Framework for Call Graph Construction Algorithms," *ACM Trans. Programming Languages and Systems,* vol. 23, no. 6, pp. 685-746, Nov. 2001.
[30] J. Whaley and M. Lam, "An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages," *Proc. Static Analysis Symp.,* 2002.
[31] O. Lhoták and L. Hendren, "Scaling Java Points-To Analysis Using Spark," *Proc. Int'l Conf. Compiler Construction,* pp. 153-169, 2003.
[32] J. Dean, D. Grove, and C. Chambers, "Optimizations of Object-Oriented Programs Using Static Class Hierarchy Analysis," *Proc. European Conf. Object-Oriented Programming,* pp. 77-101, 1995.
[33] O. Shivers, "Control-Flow Analysis of Higher-Order Languages," PhD dissertation, Carnegie Mellon Univ. 1991.
[34] L. Andersen, "Program Analysis and Specialization for the C Programming Language," PhD dissertation, DIKU, Univ. of Copenhagen 1994.
[35] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification,* second ed. Addison-Wesley, 2000.
[36] A. Rountev, "Dataflow Analysis of Software Fragments," PhD dissertation, Rutgers Univ., Aug. 2002, available as Techical Report DCS-TR-501.
[37] A. Rountev, A. Milanova, and B.G. Ryder, "Fragment Class Analysis for Testing of Polymorphism in Java Software," Technical Report OSU-CISRC-1/04-TR04, Dept. of Computer Science and Eng., Ohio State Univ., Jan. 2004.
[38] N. Fenton, S. Pfleeger, and R. Glass, "Science and Substance: A Challenge to Software Engineers," *IEEE Software,* vol. 11, no. 4, pp. 86-95, July 1994.
[39] S. Siegel and N.J. Castellan, *Nonparametric Statistics for the Behavioral Sciences,* second ed. McGraw-Hill 1988.

[40] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction.* Kluwer Academic, 2000.

[41] R. Chatterjee and B.G. Ryder, "Data-Flow-Based Testing of Object-Oriented Libraries," Technical Report DCS-TR-433, Rutgers Univ., Apr. 2001.

[42] F. Tip, P. Sweeney, C. Laffra, A. Eisma, and D. Streeter, "Practical Extraction Techniques for Java," *ACM Trans. Programming Languages and Systems,* vol. 24, no. 6, pp. 625-666, 2002.

[43] P. Sweeney and F. Tip, "Extracting Library-Based Object-Oriented Applications," *Proc. Symp. Foundations of Software Eng.,* pp. 98-107, 2000.

[44] M.J. Harrold and G. Rothermel, "Performing Data Flow Testing on Classes," *Proc. Symp. Foundations of Software Eng.,* pp. 154-163, 1994.

[45] M. Mock, M. Das, C. Chambers, and S. Eggers, "Dynamic Points-To Sets," *Proc. Workshop Program Analysis for Software Tools and Eng.,* pp. 66-72, 2001.

[46] D. Liang, M. Pennings, and M.J. Harrold, "Evaluating the Precision of Static Reference Analysis Using Profiling," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 22-32, 2002.

**Atanas Rountev** received the PhD degree in computer science from Rutgers University. He is an assistant professor in the Department of Computer Science and Engineering at Ohio State University. His research interests include program analysis, software testing, program understanding and evolution, and software tools. His recent research focuses on analysis of object-oriented languages and on tool support for reverse engineering. He is a member of the IEEE Computer Society, ACM, SIGSOFT, and SIGPLAN.

**Ana Milanova** received the PhD degree in computer science from Rutgers University in 2003. She is currently an assistant professor in the Department of Computer Science at Rensselaer Polytechnic Institute. Her research interests focus on static and dynamic program analysis and its applications in software productivity tools and optimizing compilers. She is a member of the IEEE Computer Society, ACM, SIGSOFT, and SIGPLAN.

**Barbara G. Ryder** is a professor of computer science at Rutgers University, New Brunswick, New Jersey. Dr. Ryder became a fellow of the ACM in 1998 and was selected as a CRA-W Distinguished Professor in 2004. She was selected for the Professor of the Year Award for Excellence in Teaching by the Computer Science Graduate Students Society of Rutgers University in 2003 and received the ACM SIGPLAN Distinguished Service Award in 2001. She was the general chair of the 2003 Federated Computing Research Conference and served on the Board of Directors of the Computing Research Association (CRA) from 1998-2001. She was elected a member of the ACM Council in 2000 and served on the ACM SIGPLAN Executive Committee from 1989-1999 (as SIGPLAN Chair, 1995-1997). She was a recipient of a National Science Foundation Faculty Award for Women Scientists and Engineers (1991-1996). Dr. Ryder's research focuses on static and dynamic program analyses for object-oriented languages and practical software tools. Applications include: change impact analysis, program understanding, software testing, and testing availability of web services (http://www.cs.rutgers.edu/~ryder). She is a member of the IEEE Computer Society, ACM, SIGSOFT, and SIGPLAN.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.