Class Analysis for Testing of Polymorphism in Java Software

Atanas Rountev   Ana Milanova   Barbara G. Ryder
Rutgers University, New Brunswick, NJ 08903, USA
{rountev,milanova,ryder}@cs.rutgers.edu

Abstract

Several coverage criteria have been proposed for testing of polymorphic interactions in object-oriented software. We have built a prototype tool that supports these criteria. This paper describes the overall tool design, and shows how class analysis is used by the tool to compute the coverage requirements. We discuss the importance of analysis precision and the ability to analyze subsets of partial programs. We also show how to modify Rapid Type Analysis [2] to compute the coverage requirements, and present initial empirical evaluation of analysis precision.

1 Introduction

Testing of object-oriented software presents new challenges due to features such as inheritance, polymorphism, dynamic binding, and object state [3]. Programs contain complex interactions between sets of collaborating objects from different classes; therefore, testing the interactions between classes is of critical importance. These interactions are greatly complicated by object-oriented features such as polymorphism. *Polymorphism* is the ability to bind 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 [3].

Various techniques for testing of polymorphic relationships have been proposed in the testing literature [8, 7, 9, 5, 4, 1]. Most approaches require testing that exercises all possible bindings of certain polymorphic references. Such requirements can be represented as *coverage criteria*. For example, consider the virtual call `o.m()` in Java. The requirements of several approaches [8, 7, 9, 5, 4] can be represented as a coverage criterion that requires testing of this call for every possible class of the receiver objects that may be bound to reference `o`. We refer to this as the **receiver-classes** criterion. Similar coverage requirements have also been defined for the classes of the senders and the parameters of a message [8, 9, 5].

Effective, efficient, and repeatable testing requires systems that automate the testing activities. Without such systems, testing based on coverage criteria is essentially impossible. A system that supports a particular coverage criterion (1) analyzes the software to determine what program elements need to be covered, (2) inserts instrumentation that allows coverage tracking, (3) executes the test cases, and (4) reports the degree of coverage and the elements that have not been covered. Human testers use the coverage information to estimate the adequacy of the existing test cases and to construct new test cases for exercising uncovered elements.

We are in the process of building a prototype system for testing of polymorphic interactions in Java software. Our goal is to build a tool that allows us to experiment with some of the proposed approaches for such testing (e.g., [8, 7, 9, 5, 4, 1]). By designing, implementing, and using this tool, we expect to gain first-hand experience and insights about the problems of building testing systems with support for testing of polymorphism. Identifying and solving such problems is a first step toward incorporating this form of testing in real-world testing tools.

The tool analyzes the tested software, inserts instrumentation, executes the available test cases, and reports the achieved coverage. The goal of the analysis phase is to determine the possible bindings of certain object references and to produce the corresponding coverage requirements. This goal can be achieved through *class analysis*. Class analysis determines the classes of all objects that may be bound to an object reference. Previous testing work uses the simplest form of class analysis, by considering all subclasses of the declared type of a reference. However, there is a wide variety of more precise class analyses.[1] One of our goals is to investigate the use of these analyses for the purposes of testing of polymorphism. In this context, the usefulness of the analyses is directly related to two important issues: *analysis imprecision* and *analysis of partial programs*.

Analysis Imprecision. We believe that analysis imprecision is a critical issue for making class analyses useful in real-world testing tools. Less precise analyses

[1] For example, see [6] for an overview of several class analyses.

compute less precise coverage criteria—in other words, some of the requirements may be impossible to achieve. For example, a less precise analysis may produce a large set of possible receiver classes at a virtual call, while in reality only a small subset of these classes is actually possible. Thus, regardless of the testing effort, high coverage can never be achieved. In this case the coverage metrics become meaningless: is the low coverage due to inadequate testing, or is it due to imprecise analysis? Clearly, this problem seriously compromises the usefulness of the coverage criteria.

Coverage criteria also guide the selection of new test cases for exercising elements that have not yet been covered. If a criterion is imprecise, the person that 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. The cost of such imprecision is unacceptable, because *human time and attention are much more expensive than computing power.*

One goal of this project is to investigate the degree of imprecision and the sources of such imprecision for a variety of class analyses. Such investigations are particularly important for analyses used in testing tools: they provide critical evaluation of the real-world applicability of the analyses, as well as guidelines for more precise analyses that avoid specific sources of imprecision.

Partial Programs. Class analyses are typically designed as *whole-program analyses* that process complete programs. This approach is appropriate for whole-program optimizing compilers, which are the traditional application domain of these analyses. However, the coverage criteria that we investigate are not used at the level of whole programs, but rather at smaller testing scopes (e.g., individual classes or modules). Furthermore, such testing is typically done when the program is only partially developed. Classes and modules are often tested before their clients have been implemented (e.g., during bottom-up development and testing [4]). Another typical example is the testing of reusable modules (e.g., libraries), which are specifically designed to be used by a variety of unknown clients. Existing whole-program analyses cannot be used directly in the context of such testing, and need to be modified to handle partial programs.

Goals. This paper describes our initial experience in building and using the tool. Our first goal is to present the overall tool design and implementation, as discussed in Section 2. The design allows testing of subsets of partial programs, and provides tool users with great flexibility in defining which parts of the software need to be tested and tracked for coverage.

Our second goal is to investigate the use of class analysis in the context of the tool, as discussed in Section 3. In particular, we focus on the modifications that must be made to traditional whole-program class analysis. Essentially, we need analyses that can model the behavior of unknown client code. We also show how to modify Rapid Type Analysis (RTA) [2] to solve this problem. This modified version of RTA is the starting point in our investigation of different class analyses.

Our last goal is to present the results from the initial experiments with the tool. These experiments evaluated the imprecision of the modified RTA, by determining the best possible coverage achievable with actual test cases. The difference between the required and the actual coverage is due to analysis imprecision. Our results, presented in Section 4, suggest that RTA can determine (almost) precisely what methods are actually executable. The results also indicate that RTA produces spurious receiver classes at virtual calls. We discuss some of the sources of imprecision, and present suggestions for more precise analyses that could avoid this imprecision.

2 Design and Implementation

The tool design is based on several observations. The first observation is that testing is typically done while the tested program is being built. Therefore, the tool should be designed to operate on programs that are only partially implemented. More specifically, we assume that the tool will operate on a set of classes that are stable enough to be executed, but generally do not form a complete program.

Example 1. Consider the two packages in Figure 1. Suppose that Alice designs, implements, and tests package `stations`. At a later date, Bob creates package `links`. Once he has designed the package interface[2] and has implemented the package classes, Bob starts testing his code using our tool. Of course, at this point the existing classes do not form a complete program.

The second observation is that tool users are likely to apply the coverage criteria only to a subset of the existing classes. In our example, Bob would typically try to achieve coverage for his own code (e.g., for the two virtual calls to `postRequest` in `links`), but not for Alice's code in `stations`. Thus, the tool allows the user to define a *component under test* (CUT),[3] which is a set of classes in which coverage should be tracked. The tool inserts instrumentation that only tracks coverage for statements inside the CUT. Typically, the CUT includes the classes whose functionality is the primary

[2] Here and for the rest of the paper, we use "interface" to refer to the software engineering concept of an interface, not the `interface` construct in the Java language. For our purposes, a Java `interface` is considered a special case of an abstract Java class.

[3] Here we use "component" in the generic sense, not in the sense used in the context of component-based software.

```
package stations;
public abstract class Station
   { public abstract void postRequest(String req); }
public class SmallStation extends Station { ...  }
public class CentralStation extends Station { ...  }

package links;
import stations.Station;
public abstract class Link {
   public abstract void sendRequest(String req);
   Link(Station s) { st = s; }
   Station st;
   static long req_id = 0;
}
public class DefaultLink extends Link {
   public DefaultLink(Station s) { super(s); }
   public void sendRequest(String req)
      { st.postRequest(req_id++ + " " + req); }
 }
public class LinkFactory {
   public static Link getPriorityLink(Station s)
      { return new UrgentLink(s); }
}
class UrgentLink extends Link {
   UrgentLink(Station s) { super(s); }
   public void sendRequest(String req)
      { st.postRequest("URG " + req_id++ + " " + req); }
 }
```

Figure 1: Packages `stations` and `links`.

focus of the testing, and for which the user has enough understanding to achieve good coverage. In our example, Bob would define the CUT to be package `links`. In general, the CUT could be an arbitrary set of classes.

Testing Tasks. The goal of the tool user is to test certain functionality provided by the CUT. The interface to this functionality is defined by a set of *relevant methods* and *relevant fields*. Any client code can access the functionality by referencing these relevant methods and fields. The testing itself is done through test cases that exercise this interface, in a manner that essentially simulates the possible uses of the functionality by arbitrary client code.

Example 2. Consider the functionality of sending requests via default links using the packages from Figure 1. Client code that uses this functionality creates instances of `DefaultLink` and invokes `sendRequest` on them. In addition, such code creates objects of classes `SmallStation` or `CentralStation` (in order to instantiate `DefaultLink`), and possibly invokes on them some of the public methods from `stations`. Therefore, the relevant methods for this functionality are the constructor of `DefaultLink`, `DefaultLink.sendRequest`, and certain methods from `stations` (constructors, etc.).

One of our goals is to allow flexibility in defining the targets of the testing. Thus, we assume that tool users define and perform *testing tasks*, where each task targets a specific functionality of interest. For each task, the user provides a *task definition* which contains a list of all relevant methods and fields for the tested functionality. The goal of the testing task is to exercise these methods and fields in a manner that models the behavior of any possible client code.

A field is relevant if it must be referenced in order to access the *entire* tested functionality; in other words, without referencing that field, some aspects of the functionality cannot be accessed. Similarly, a method is relevant if, in order to access the entire functionality, client code must contain call statements for which the *static target* of the call is that method. (An example below illustrates the need to consider the static targets of virtual calls instead of the actual run-time targets).

Relevant methods and fields can be declared both inside and outside of the CUT. For example, some methods in `stations` (e.g., constructors) are relevant with respect to the functionality provided by default links, even though they are not in the CUT.

A relevant method can be an abstract method. For example, consider client code that sends requests via the urgent links from Figure 1. Since `UrgentLink` is not public and is not visible outside of `links`, no client code can directly reference `UrgentLink.sendRequest`. All external references to instances of `UrgentLink` (obtained through the factory class) have declared type `Link`. Therefore, calls to `sendRequest` through such references have static target `Link.sendRequest`. Thus, `Link.sendRequest` is a relevant method for any task that tests urgent links. We observed many examples of this situation during our experiments.

Once the tester has defined a testing task, she uses the tool to compute testing requirements with respect to this task and with respect to several coverage criteria. The user can choose coverage for (i) all possible classes of message receivers (i.e., the receiver objects at virtual calls), (ii) all possible classes of message senders, (iii) all possible classes of message parameters, or (iv) any combination of the above. This approach allows the implementation of several coverage requirements proposed in previous work [8, 7, 9, 5, 4]. For brevity, in this paper we only discuss the **receiver-classes** criterion.

The above criteria require information about the possible bindings of object references. Such information can be computed through class analysis; however, traditional whole-program class analysis cannot be used in this context. We need an analysis that takes into account all possible uses of relevant methods and fields by arbitrary client code. Section 3 discusses this issue and

```
package harness;
public abstract class TestCase
    { public abstract void run(); }
package linktests;
import links.DefaultLink;
import stations.SmallStation;
public class DefaultLinkTest extends harness.TestCase {
   public void run() {
      SmallStation s = new SmallStation();
      DefaultLink l = new DefaultLink(s);
      l.sendRequest("start");
      // send more requests, check results, etc.
   }
}
```

Figure 2: Simplified test case.

shows how to modify RTA to solve this problem.

Given the computed requirements, the tool inserts instrumentation that allows tracking of these requirements. For example, the inserted code allows every execution of a virtual call to be recorded at run time; the classes of the actual receiver objects are determined through the Java reflection mechanism. Both the analysis component and the instrumentation component of the tool are implemented using the Soot framework (`www.sable.mcgill.ca`) [10].

The instrumented code is supplied to a *test harness* which automatically executes test cases for this code. The harness is based on harness designs from [4] and from the Mauve project (`sources.redhat.com/mauve`). Each test case is a Java class. Figure 2 shows a simplified test case for the task in Example 2.[4] The harness loads each such class from a given list, creates an object of this class, and invokes the `run` method for this object. After all test cases have been executed, the coverage results are reported to the user. The harness also provides details about calls for which 100% coverage was not achieved; this information can then be used by the user to create additional test cases.

3 Class Analysis

The goal of a testing task is to exercise the entire tested functionality in a manner that essentially simulates the behavior of all possible clients of the functionality. Correspondingly, the goal of the analysis should be to determine all possible reference bindings under the assumption that arbitrary unknown client code can access the relevant methods and fields. Therefore, we define the following analysis problem: for each object reference that occurs in the CUT, compute the set of all possible classes, assuming that (1) unknown client code accesses all relevant fields, and (2) for each relevant method, unknown client code contains a call whose static target is that method. This problem requires class analysis that essentially works under worst-case assumptions about the clients of the tested functionality.

Example 3. Consider the testing task for default links from Example 2. The relevant methods are the constructor of `DefaultLink`, `DefaultLink.sendRequest`, and some additional methods from `stations` (e.g., constructors). Code that accesses relevant methods may create instances of `SmallStation` or `CentralStation` and then assign these instances to field `st` (by calling the constructor of `DefaultLink`). The analysis should determine these two possibilities for `st`, and should construct the coverage requirements accordingly.

The analysis should take into account *only* relevant methods and fields. For example, consider a testing task for links to small stations. The constructor of `CentralStation` is not relevant for this task, and the analysis should assume that this constructor is *not* accessed by unknown client code. Otherwise, the analysis would have to conclude that `st` may refer to instances of `CentralStation`, which would impose testing requirements that are beyond the purpose of this task.

Another important observation is that the analysis should take into account *only* classes that exist at the time of the testing. For example, future client code may create new subclasses of `Station`. In general, field `st` may refer to instances of these new subclasses. However, at the time of the testing, these classes do not exist and cannot be included in the testing requirements. Furthermore, the code in such future classes cannot be executed at the time of the testing, and therefore the effects of such code should be ignored. For example, the run-time targets of the calls through `st` may be methods in `SmallStation` and `CentralStation`, as well as methods in future subclasses of `Station`. The analysis should take into account the effects of the two existing target methods from `stations` (by analyzing their bodies, and recursively, all of their callees). However, the analysis should *not* attempt to model the possible effects of non-existent target methods declared in future subclasses, because these effects cannot be observed at the time of the testing.

We have modified Rapid Type Analysis (RTA) [2] to solve the above analysis problem. RTA is a popular whole-program class analysis. Our initial implementation of the tool uses a modified version of RTA to compute the possible bindings of object references. This analysis is the starting point in our investigation of different class analyses in the context of the tool; additional analyses will be implemented and evaluated in

[4]Note that this test case does not satisfy the **receiver-classes** criterion because the call to `postRequest` in `DefaultLink.sendRequest` is not exercised for the `CentralStation` receiver class.

our future work.

The standard RTA 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 reference `o` is the set of all instantiated subclasses of the declared type of `o`. For every processed non-virtual call, the target method is added to the worklist of reachable methods. When a virtual call `o.m()` is processed, the analysis considers each non-abstract subclass C of the declared type of `o`. If C is instantiated, the appropriate run-time target method becomes reachable. If C is not instantiated, the run-time target is placed in a pending set associated with C. If later C becomes instantiated, all methods in the pending set for C become reachable.

Our modified version of RTA starts by processing the list of relevant methods from the testing task definition; after this preprocessing, the analysis works exactly like the standard RTA. Intuitively, the goal of the preprocessing phase is to simulate the possible effects of the unknown client code. For every relevant constructor, the corresponding class is added to the set of instantiated classes. Every relevant non-virtual method (including constructors) is immediately added to the worklist of reachable methods. For every relevant virtual method m declared in class C_i, the analysis considers all non-abstract subclasses of C_i. For each such subclass C_k, the analysis searches C_k and all of its superclasses for the first method m' that either overrides m or is m itself. If C_k is already instantiated, m' is made reachable; otherwise, m' is added to the pending list for C_k.

Example 4. Consider a task for testing the urgent links from Figure 1. As discussed in Section 2, one of the relevant methods for this task is `Link.sendRequest`. Suppose that this is the first method processed during the preprocessing phase. Since the subclasses of `Link` have not been instantiated, `DefaultLink.sendRequest` is added to the pending list for class `DefaultLink`, and `UrgentLink.sendRequest` is added to the pending list for class `UrgentLink`. Since `getPriorityLink` is also a relevant method, it becomes reachable during the preprocessing phase. When later the body of this method is processed, class `UrgentLink` becomes instantiated and `UrgentLink.sendRequest` becomes reachable. In turn, this makes reachable the two `postRequest` methods in `stations` (and recursively, all of their callees).

Classes `SmallStation` and `CentralStation` are instantiated during the preprocessing phase because their constructors are relevant. The solution for field `st` contains both of these classes. The only reachable call through `st` is in `UrgentLink.sendRequest`. Thus, the computed coverage criterion requires testing of this call for each of the two possible receiver classes.

4 Empirical Results

Our initial experiments had two goals. First, we evaluated the imprecision of RTA. As discussed in Section 1, analysis imprecision is a critical issue for the usefulness of any class analysis in the context of testing tools. Our second goal was to determine the sources of analysis imprecision, in order to provide insights for more precise analyses that could avoid this imprecision.

We used our modified version of RTA to compute the requirements of the **receiver-classes** criterion. Then we tried to achieve the maximal possible coverage of these requirements by writing and executing actual test cases.[5] Requirements that could *not* be satisfied were due to analysis imprecision. Such spurious requirements seriously compromise the usefulness of any testing tool.

We defined three testing tasks for exercising some of the functionality provided by the Java library packages `java.text` and `java.util.zip` from JDK 1.1.5. For example, task *zip* tested the reading of ZIP archives, which is implemented by several classes from package `java.util.zip`. Another task *format* tested the formatting of numbers, dates, and messages through classes from `java.text`. The first two columns of Table 1 shows the number of CUT classes, as well as the number of *relevant classes* (i.e., classes that contain relevant methods or fields).[6]

We first considered the set of RTA-reachable methods in the CUT; only calls in such methods are included in the **receiver-classes** criterion. If any of these methods is actually unreachable, it could introduce spurious testing requirements. Furthermore, unreachable methods also present a problem for other coverage criteria (e.g., traditional statement coverage). To determine actual reachability, we inserted additional instrumentation and augmented the test cases to achieve maximal method coverage. Part (b) of Table 1 shows the number of RTA-reachable methods and how many of them were actually unreachable. The results suggest that unreachable methods will not be a serious problem for RTA or for more precise analyses.

Next we considered all virtual calls (in the CUT) for which RTA reported more than one possible class of the receiver. Since in many cases there were multiple calls through the same variable, we partitioned the calls into equivalence classes. Each equivalence class contains virtual calls through the same variable and with the same

[5] We made substantial effort to ensure maximal coverage (even though it cannot be formally proven that we achieved it)—working independently, two of us wrote test cases to achieve the best possible coverage, and then the results were compared and corrected.

[6] Recall that (i) relevant methods and fields need not be in the CUT, and (ii) not all CUT classes are relevant (e.g., class `UrgentLink`).

Task	(a) Classes		(b) Methods		(c) Polymorphic Call Sites							
	CUT	Relevant	RTA	Unreach	Sites	EqClasses	$\delta = 0$	$\delta = 1$	$\delta = 2$	$\delta = 3$	$\delta = 4$	$\delta = 5$
zip	8	6	43	0	5	2	–	1	1	–	–	–
iter	12	5	73	0	12	1	1	–	–	–	–	–
format	13	27	226	2	79	35	12	2	1	14	1	5

Table 1: Task description and coverage results.

actual set of receiver classes in the maximal coverage. The number of polymorphic calls and the number of equivalence classes are shown in the first two columns of part (c) in Table 1. Each of the remaining columns in part (c) corresponds to a specific number δ of receiver classes that were reported by RTA but were actually impossible to achieve in the maximal coverage; thus, δ is a metric of imprecision. Each column shows the number of equivalence classes with that particular value of δ. For example, for the single equivalence class in *iter*, RTA computes a precise solution (i.e., $\delta = 0$). On the other hand, for most equivalence classes in the other tasks, RTA produces at least one spurious receiver class.

These initial results indicate that RTA tends to produce spurious receiver classes. While writing the test cases, we had the opportunity to observe the sources of this imprecision. The most important source is the inability of RTA to track the flow of reference values. Reference values are produced by `new` statements and are propagated through series of assignments, parameter passing, and return statements. Based on our observations of the actual tested code, we believe that significant precision improvements will be achieved by analyses that track the flow of reference values. Another source of imprecision is the inability to take into account conditions of the form "`if (x instanceof Y)`". In several cases, we observed regions of code guarded by such conditions, with the corresponding restrictions for the receiver classes inside such regions.

Clearly, more extensive experiments are needed to confirm our initial observations. However, the initial results suggest that more precise class analyses are needed for the purpose of testing of polymorphism. A wide variety of existing class analyses can provide better precision that RTA. It would be particularly interesting to investigate analyses that track the flow of reference values, and to determine empirically the appropriate degree of precision for this tracking. In addition, it would be interesting to investigate analyses that take advantage of `instanceof` conditions, possibly by employing some limited form of intraprocedural flow-sensitivity.

5 Future Work

Our initial results need to be confirmed by more extensive experiments with RTA. In addition, similar experiments are needed for other, more precise class analyses. We plan to investigate several existing whole-program class analyses in the context of the tool. Such analyses would have to be modified to analyze partial programs with respect to a given testing task, as discussed in Section 3. This problem raises interesting analysis issues.

There are two other promising directions of future work. First, it would be interesting to add tool support for more coverage criteria (e.g., for coverage of def-use relationships involving polymorphic references). In addition, it would be valuable to augment the tool to assist the users in writing test cases. Such functionality could be of great help in reducing the time and effort spent by human testers.

Acknowledgments. This research was supported by NSF grants CCR-9804065 and CCR-9900988.

References

[1] R. Alexander and J. Offutt. Criteria for testing polymorphic relationships. In *International Symposium on Software Reliability Engineering*, 2000.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, 2000.