

Putting Escape Analysis to Work for Software Testing

Amie L. Souter
Department of Computer Science
Drexel University
Philadelphia, PA 19104
souter@mcs.drexel.edu

Lori L. Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

Abstract

Developed primarily for optimization of functional and object-oriented software, escape analysis discerns information to determine whether the lifetime of data exceeds its static scope. In this paper, we demonstrate how to apply escape analysis to software engineering tasks. In particular, we present novel software testing and retesting techniques for object-oriented software which utilize escape analysis. We exploit a combined pointer and escape analysis that is able to identify how individual objects allocated in one region of a program interact with other regions of a program. The analysis framework increases flexibility and scalability as testing coverage can be targeted to a specific arbitrary region of a program, followed by integration testing that can be focused on particular sets of objects escaping the region. We demonstrate how regression testing can be performed utilizing this framework. We believe such a flexible framework becomes increasingly beneficial as applications become more component-oriented.

1. Introduction

As the Internet continues to become more important in everyday life, web-based applications will be required to be more robust. Web-based programs are often written in many different languages and consist of many different interacting units. Therefore, analysis techniques that are able to analyze arbitrary regions of interacting code will be beneficial for various software engineering tasks, including regression testing and software maintenance activities such as impact analysis and program understanding.

While escape analysis was invented for compiler optimization, it can be exploited for software engineering tasks. Escape analysis provides information that determines whether the lifetime of data exceeds its static scope. In an object-oriented program, an object o is said to *escape* a method m of the program if the lifetime of o may exceed

the lifetime of m [6]. This notion can be generalized to characterize how objects allocated in one region of a program can escape to be accessed by another region [27]. This paper demonstrates how knowledge about the interaction of objects between different regions of an object-oriented program can provide information useful for software maintenance and enable novel integration and regression testing techniques.

Over the years, the focus of program analysis techniques has changed from single procedures in isolation to whole programs to the more recent focus on incomplete programs including libraries and client software without access to the called library code. Whether the analysis unit is a procedure, whole program, or program fragment, the analyses have historically viewed the unit of analysis in isolation of its potential context, and performed analysis with conservative assumptions about the potential context of the unit. The primary goal of most analyses has been to compute the most precise results possible for that unit within a practical analysis time.

Similarly, current testing frameworks perform levels of testing starting with unit testing, followed by integration testing, and then system testing. The unit for testing object-oriented programs is typically a class. The overall goal of this multi-level testing framework is to identify different types of failures in a systematic manner. Each testing level aims to reveal the problems inherent in the application unit or interface level being targeted.

By utilizing escape analysis results, we can relax the framework for software testing to allow arbitrary regions as the units for targeted test coverage and then integration testing on an object basis. Escape analysis is unique as an analysis technique as it provides valuable information about the interaction *between* the current analysis region and its surrounding context. In particular, Whaley and Rinard [27] developed an escape analysis for object-oriented software that is capable of analyzing arbitrary regions of complete or incomplete programs, and provides an abstraction that represents interactions between analyzed and unanalyzed re-

gions of a program.

Specifically, in this paper, we show how to exploit escape analysis to develop a novel approach to testing and retesting in response to changes which has the following key properties:

- Test coverage can be targeted to arbitrary regions of the program, not necessarily a single method, single class, or single package.
- Testing is object-based, providing test tuples for the manipulation of objects created in the unit, with knowledge and feedback for manipulations of objects in the unit that have possible interactions with other units.
- Integration testing follows unit testing on arbitrary regions allowing one to focus on particular sets of objects that escape the region if desired. Thus, integration testing can be targeted to particular objects and their manipulation.

A testing framework with these properties gives the tester increased flexibility and scalability in testing. Flexibility is provided in both the unit of testing and the focus of integration testing. In addition, the tester can establish priorities for integration testing when testing resources are limited. The testing process can be made more scalable by focusing testing efforts on particular object interactions, determined by static program analysis. This technique is meant to complement black-box testing based on UML or other object specifications, as this technique gives a finer granularity of object manipulation for testing coverage.

The remainder of this paper is organized as follows. Section 2 presents an overview of escape analysis with a description of the Whaley and Rinard [27] points-to escape graph model. Section 3 presents our object-based integration testing approach, beginning with an overview of our previous work on contextual object-based testing techniques for unit testing [24]. Section 4 presents algorithms to demonstrate the ease of regression testing in this framework. We present results from our empirical study to evaluate the potential effectiveness of escape analysis for testing in Section 5. Finally, conclusions and future work are given in Section 6.

2. Escape Analysis

Escape analysis was originally developed to deduce the lifetime of dynamically (and implicitly) created objects in functional programs in order to improve automatic memory management [9, 8, 3]. More recently, escape analysis has been used for optimizing object-oriented programs [27, 6, 4, 5, 26, 23]. By identifying the objects that do not escape each method, the compiler can allocate those

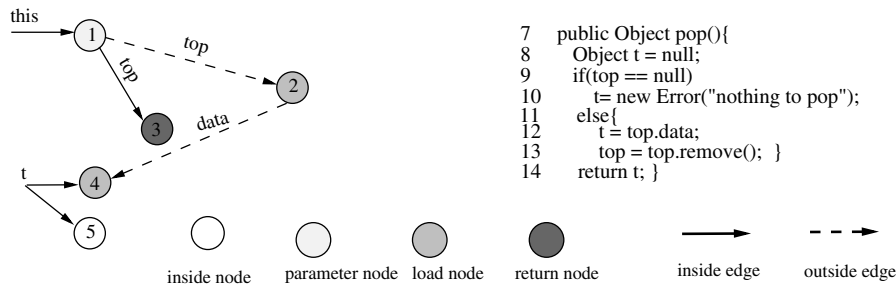
objects on the stack and reduce the overhead of heap allocation and deallocation. In addition, safe elimination of unnecessary thread synchronization operations is achieved by determining which objects are isolated to each thread.

A number of different techniques for escape analysis in object-oriented programs have been developed [27, 6, 4, 5, 26, 23]. We utilize the combined pointer and escape analysis approach presented by Whaley and Rinard [27] due to its compositional nature and ability to handle incomplete programs. Their analysis is based on building a program representation that combines points-to information about objects (i.e., points-to graphs) with information about which object creations and references occur within the current analysis region (the set of methods that are currently being analyzed) versus outside the analyzed region of a program. The points-to graph for each method characterizes how local variables and fields in objects refer to other objects. The escape information can be used to determine how objects allocated in one region of the program can escape and be accessed by another region of the program. Their analysis is able to analyze incomplete program units as well as arbitrary parts of the program by establishing the current analysis region, and providing complete information about objects that do not escape the analyzed region.

In a points-to escape graph, nodes represent objects that the program manipulates and edges represent references between objects. Each kind of object that can be manipulated by a program is represented by a different kind of node in the points-to escape graph. Figure 1 presents the complete points-to escape graph for a single method. For this example, the current analysis region is the method `pop` only. Each *inside node* represents an object creation site for objects created and reached by references created inside the current analysis region of the program. In Figure 1, the inside node 5 represents an instance of the `Error` class referenced by `t`.

Outside nodes represent objects created outside the current analysis region or accessed via references created outside the current analysis region. There are several different kinds of outside nodes: parameter nodes, load nodes, and return nodes. There is one *parameter node* for each formal parameter of the method; a parameter node represents the object that its parameter references during the execution of the analyzed method. Note that the receiver object is represented as the first parameter of each method. Node 1 referenced by `this` in Figure 1 illustrates an outside (parameter) node.

Each access to an object field `f` of an object `v` is translated into the form `v.f` and called a load statement. Each load statement in the program has a corresponding *load node* that represents all of the outside objects whose references are loaded at the given load statement, if the loaded reference could indeed be to an outside object. The pred-



```

7 public Object pop(){
8   Object t = null;
9   if(top == null)
10    t = new Error("nothing to pop");
11  else{
12    t = top.data;
13    top = top.remove(); }
14  return t; }

```

Figure 1. Example of complete points-to escape graph for a single method.

icate `top == null` generates the load node 2 referenced by `top`. Finally, there is one *return node* for each method invocation site in the method to represent the return values of the method invocation of unanalyzed methods. In Figure 1, we assume that method `remove` is outside the current analysis region, and thus the reference `top` points to the return node 3 for the call to `remove`. A return value is different than a return node. The return value represents the return statement while the reference representing the return value points to all nodes that could be returned at the end of the method. In Figure 1, `t` points to the possible return values, represented by nodes 4 and 5.

There are also two different kinds of edges. An *inside edge* represents references created inside the current analysis region. Inside edges from outside nodes or nodes reachable from outside nodes represent the situation in which the unanalyzed region *may read* a reference created inside the current analysis region. An *outside edge* represents references created outside the current analysis region. Outside edges represent the situation in which the current analysis region *reads* a reference created in an unanalyzed region. Thus, each outside edge points to a load node, as shown in Figure 1, where the references `top` and `data` both point to load nodes 2 and 4, respectively.

The distinction between inside and outside nodes and edges is important because it is used to characterize nodes as either captured or escaped. *Captured* nodes correspond to the fact that the objects they represent have no interactions with unanalyzed regions of the program, and the edges in the graph completely characterize the points-to information between objects represented by these nodes. On the other hand, *escaped* nodes represent the fact that the objects represented by them escape to unanalyzed portions of the program. An object can escape in several ways: (1) A reference to the object is passed as a parameter to the current method. (2) A reference to the object is written into a static class variable. (3) A reference is passed as a parameter to an invoked method and there is no information about the invoked method. (4) The object is returned as the return value of the current method. Figure 1 illustrates an example of an escaped object. The return value nodes 4 and 5 refer-

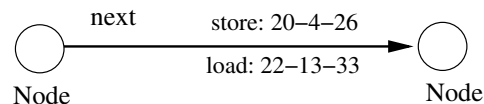


Figure 2. Illustration of ape graph annotation.

enced by `t` escape the current analysis region, which is the method `pop`. Even though an object escapes the method, it may be recaptured by a caller of the method.

The points-to escape graph is built in a flow sensitive manner for a given method. At each statement `s` in the method, the points-to escape graphs representing each of the predecessor statements of `s` in the control flow graph are first merged into a single graph. The new points-to escape graph in effect at the program point immediately after a statement `s` is constructed by applying the effects of `s` to the merged graph that was in effect immediately before `s`.

When a method call is encountered during the construction of a points-to escape graph for a given method, interprocedural analysis is performed. Interprocedural analysis is achieved through merging the parameterized points-to escape graphs of all the potentially invoked methods at the call site with the points-to escape graph at the point immediately before the call site, to form the points-to escape graph at the point just after the call site. For call sites to unanalyzed methods, the parameters and return value are marked as escaped within the caller's graph.

A key feature of this analysis is its compositional nature, meaning that each method is analyzed independently of its callers. In addition, a method can be analyzed independently of any methods that it invokes. The analysis provides information about where (source line number) these objects escape and how they escape in terms of the type of node that is associated with them.

3. Object-Based Integration Testing

While unit testing focuses on faults local to a unit, integration testing has the goal of exercising the interactions

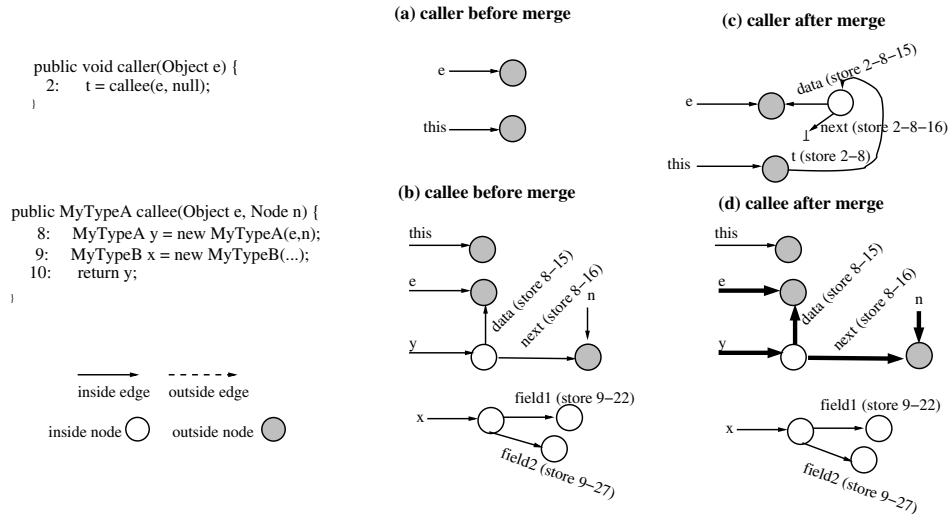


Figure 3. Interprocedural merge example.

between different units, often developed by different programmers. For object-oriented software, the basic unit is typically a class, and integration testing is aimed at determining whether instances of classes related in some way (via inheritance, aggregation,...) interact as expected.

Historically, a fundamental decision in integration testing has been the order in which different units are integrated and tested. One strategy is to compute the dependences between different classes, and then integrate classes in a top-down or bottom-up manner based on the dependences, with some special handling for cyclic dependences among classes [16, 17, 13]. Other strategies include threads integration, critical modules integration, big-bang integration, and incremental integration [12, 15, 16].

Orso [16] addresses the problem of adequately selecting test cases for testing combinations of polymorphic calls during integration testing. Based on data flow testing, the approach extends the traditional definitions of def and use to take into account polymorphism. The new definitions are also used to extend the traditional data flow coverage criteria [18], therefore allowing for the selection of execution paths that might reveal failures due to incorrect combinations of polymorphic calls. Orso's technique is based on using the inter-class control flow graph (ICCFG), defined by Harrold and Rothermel [22]. The ICCFG represents the flow of control between interacting classes by adding edges between callers and callees of different classes' CCFGs. The nodes of the ICCFG are annotated with def and use sets. The technique assumes that the programs are alias free and calculates intra-method def-use chains. There are no results to show the amount of coverage obtained from using this approach.

In this section, we describe a novel strategy for integra-

tion testing based on escape analysis which enables testing of integrations of arbitrary regions in response to modifying portions of a code. In particular, our algorithms for interaction identification and integration test coverage exploit the escape and object manipulation information available in *annotated* points-to escape graphs defined in our previous work on object-based testing. We begin with a brief description of our approach for computing test tuples based on object manipulations of an arbitrary program region [24].

3.1. Overview of Object-Based Testing

A key component of our previous work on object-based testing is a novel formulation of definitions, uses, and def-use associations for objects in object-oriented programs [25]. The resulting object def-use associations are *contextual* in that they provide context with the computed def-use associations (in the form of partial call sequences) by exploiting the relations that occur between classes and their instantiated objects due to aggregation. By extending the escape points-to graph representation, we developed and implemented three strategies for achieving different levels of context for contextual def-use associations.

For object-based testing, we extended the points-to escape graph by adding annotations to edges in the graph. The annotations provide information about where basic object manipulations (i.e., loads and stores of objects) occur within a program. Our resulting representation is called an *Annotated Points-to Escape* (ape) graph¹. A detailed description of the ape graph representation, construction algorithm, and examples are presented in [24]. Here, we give a brief overview.

¹The ape graph apes, or mimics, the object manipulations potentially performed at run-time.

Figure 2 shows an example set of annotations on a single edge of an ape graph. The edge labeled `next` represents a reference from a field named `next` of the object of type `Node`, annotated with both a load and store annotation. The annotations indicate that there exist both a load and store of the field `next`. Further, the locations where the load and store occur are maintained through the annotations. The annotation, `(store 20-4-26)`, represents a call sequence of two method calls. The first call is at line 20 of the program followed by a second call on line 4 which can lead to a store of an object into the field `next` at line 26. Similarly, the load of the field `next` occurs at line 33, following a chain of calls from lines 22 and 13.

A contextual def-use association (cdu) can be constructed based on the annotations of the edge. The cdu `(s,(20-4-26),(22-13-33))` represents the fact that the object `s` of type `Stack` is defined by a definition of the field `next` at statement 26 (which is associated with the class `Stack` through the aggregate relation with the `Node` class), after a sequence of calls from statement 20 and then another call at statement 4, and then `s` is used after the execution of a sequence of calls (at lines 22 and then 13) that lead to a load of field `next` at line 33, and that a def-clear path between the def and the use exists.

After a fairly precise call graph is constructed for the program, we build one ape graph representation per method in the program. Nonrecursive programs can be processed in a reverse topological sort of the call graph, while recursive programs will involve fixed-point iterative analysis within each strongly connected component of the call graph. The ape graph for a method is constructed by a slight modification to Whaley and Rinard's construction algorithm, in order to build the annotations and save edges needed for testing which would be deleted by Whaley and Rinard [27]. In particular, a given load/store annotation on an ape graph edge is incrementally computed by modifying the points-to escape graph merge algorithm performed at a call site to merge a callee's ape graph into a caller's ape graph. As an edge is mapped from a callee's ape graph into the caller's ape graph at a call site in the caller, the statement number, and the earliest visible statement for the call site are concatenated onto the annotation sequences mapped from the callee's ape graph.

Because the interprocedural merge is key to understanding our annotations, contextual def-use associations, and the integration testing approach, an example of the modified interprocedural merge is shown in Figure 3. The example shows the graphs for the caller and callee before and after the method invocation. The caller's graph before the method invocation consists of only the parameters `e` and the implicit parameter `this`. The graphs after the merge operation illustrate the changes in the caller's and callee's graphs. The heavier lines in the callee's graph depict the mapping

of the callee graph edges to the caller's graph. All of these edges and nodes are marked in the callee's graph to prevent duplicate contextual def-use associations from being constructed. The incrementally constructed annotations are also illustrated.

After the annotated ape graph is constructed for the current analysis region, the contextual def-use associations are computed by a separate topological traversal of the call graph starting at the root, analyzing each call graph node's ape graph. Duplicate contextual def-use associations are avoided by not processing marked edges in an ape graph. The load and store annotations on each ape graph edge are analyzed to identify contextual def-use associations.

3.2. Arbitrary Region Analysis Scenarios

Building on our observations from our work on object-based testing, the remainder of this paper describes our new work on putting escape analysis to work for software testing tasks with particular attention to software maintenance. In this section, we describe the kinds of problems that can be addressed by analyzing the escape information provided in the ape graphs for a program region. A key characteristic of all of the problems is that the questions are posed with respect to some arbitrary region of program interest determined by the program developer, tester, or software maintainer. We call this the *region of focus*. The region could be composed of any set of classes or methods. Information can be obtained regarding interactions local to (captured within) that region, but more importantly, interactions between the region of focus and any other region. Furthermore, information can be obtained at the level of objects created within an arbitrary region. This provides more detailed information for the tester or system maintainer who needs to follow the interactions of the objects created in one region with other regions of code.

For the following descriptions, let R be the region of focus specified by the programmer, tester, or software maintainer. We refer to all code outside R and interacting with R as the *outside region*. We first present some problems involving the identification of interactions with R , and then identify some problems involving integration test coverage.

Object-based Interactions: *Identify all methods outside region R that interact with objects created at the statement S in region R .* This problem depicts situations where the user wants detailed information about the inter-region interactions of particular objects created within R . Objects may be composed of complex relationships, as well as inherit methods and attributes from parent classes. By specifying the current analysis region, and exploiting the escape analysis results for particular statements with respect to the current analysis region, we are able to obtain the set of methods in which the set of specified objects interact. This information can be used for focusing integration testing on particular

objects and interactions among specific regions of the program, in order to test a specified object in conjunction with all of its operations used in a particular context.

Region-based Interactions: *Identify all methods outside region R that interact with any object created inside R .* This problem broadens the scope of the previous problem to identify all of the interactions with the region of focus R , involving any of the objects created inside R . Thus, the larger set of all methods that interact with R through escaped objects is identified, representing a region-based problem, as opposed to an object-based problem. One example situation where this problem would be posed is in the identification of all object manipulations needed to be covered for integration testing of this arbitrary region within its many contexts.

Coupled-Regions Interactions: *Identify all methods in region T that interact with any object created inside R .* This problem examines the interactions between two specific arbitrary regions, identifying the methods in one region that interact via object manipulations with objects allocated in the other region. This is a restricted form of the region-based interactions problem, requiring distinction between regions in which methods outside the region of focus R reside. Addressing this question allows integration testing to focus on the interactions between two particular regions, at a detailed level of object manipulations and interactions.

Regional Unit Coverage: *Construct a set of test tuples (i.e., contextual def-use associations (cdus)) to cover all the object manipulations for objects created at statement S in region R , which are internal to region R .* Since classes often define complex relationships between other classes, it is difficult to unit test a single class in isolation. Often stubs or drivers are developed to simulate these complex functionalities, but it is difficult to emulate these complex interactions in a meaningful manner. Instead, we propose to put the class into its intended context, but focus first on unit testing the class by providing a way to identify the class unit test tuples within context. This is done by identifying the set of objects created at a statement S and interacting within a specified program region, where the region is the class unit. These objects are actually those found to be captured by escape analysis for region R .

Object Integration Coverage: *Construct a set of test tuples (i.e., cdus) to cover all the object manipulations for objects created at statement S , which interact with specified package P , class C , or method M outside region R .* In this situation, the outside region could be composed of code written by another developer in the same project or maybe written as part of a utility package or library. The level of integration is specified by the tester, for example, all methods in a package P , all methods within some class C , or a single specified method M . The integration level determines the size of the region in which integration testing will occur with respect to region R and statement S inside R . In this

way, the tester can incrementally test with respect to code outside, but interacting with region R . The constructed cdus represent the interaction between R and the specified region (P , C , or M) for objects created at statement S .

All Object Integration Coverage: *Construct a set of test tuples (i.e., cdus) to cover all object manipulations for objects created at statement S , which interact with any other part of the program outside region R .* In this situation, test tuples are constructed for object-based integration testing of region R with the outside region, focusing on the objects created at statement S in region R . This differs from the previous problem as all the interactions with objects at statement S are identified and cdus are constructed. Integration testing can focus on specific targeted sets of objects.

3.3 Applying Escape Analysis

This section demonstrates how escape information available in the ape graph representations of methods in an arbitrary region of focus can be applied to easily provide solutions to the problems posed in the previous section. Figure 4 contains algorithms for identifying object-based and region-based interactions for a region of focus R .

To identify all methods outside R that interact with objects created at a particular statement S in R , the `Object-based_Interactions` algorithm also takes as input the method M in which statement S resides. To compute interactions, we need the escape information for objects created at S , which is available on the ape graph nodes associated with objects created at S . In order to compute the ape graph of method M containing S , we need only compute the ape graph for M and any methods in the call subgraph of R rooted at M since those ape graphs are the only ones that could possibly be involved in the construction of the ape graph for M .

The first algorithm begins by first calculating a call subgraph of R , rooted at M . Following a reverse topological ordering of this call subgraph, an ape graph is constructed for each of the methods represented in the call subgraph, culminating in an ape graph for M . As each ape graph is constructed, an escape function that records the escape information for each node is computed. The escape information per node includes nodes escaping due to being reachable from a parameter, a static class variable, or an object passed as a parameter to or returned from a skipped method invocation site. Once the ape graph for M is constructed, we can return the escape function results pertaining to the objects created at statement S . The escape information provides the set of interacting methods. The `Region-based_Interactions` algorithm presented in Figure 4 is similar to the `Object-based_Interactions` algorithm, but the desired information pertains to *all* objects that were created inside region R . To identify the objects created inside R , we examine all of the *inside* nodes of ape graphs

for methods in R .

Figure 5 presents algorithms for two different types of testing coverage based on extending our previous work on contextual def-use associations. Given a region R , and statement S within method M in R , the `Regional_Unit_Coverage` algorithm constructs a set of cdus to cover all the object manipulations for objects created at S which are internal to R . The region R can be specified as a set of methods, methods in one particular class, a set of methods in a package, or any other arbitrary region of cooperating methods. Similar to the interactions identification algorithms, the algorithm begins by constructing the partial program representation needed to construct the cdus for R . In this case, the ape graphs for each method in the call subgraph of R rooted at method M are constructed. All of these ape graphs must be constructed in order to ensure that all interactions between the objects created at S and the rest of the region are captured. However, we need only process a subset of the edges in the ape graph for method M to construct the cdus. In particular, we compute the cdus by processing each inside edge whose source node is identified as an object that was created at S . Individual cdus are constructed by processing each store on an edge, and then finding corresponding reaching loads of the store in question. Each inside edge corresponds to a field of the object created at S , and also ensures that the object manipulation occurred inside of region R .

The `Object_Integration_Coverage` algorithm provides integration testing between two regions with respect to a particular set of objects. A set of cdus is computed to cover the objects created at statement S in region R that interact with another specified region I outside R . First, we execute the `Object-based_Interactions` algorithm to compute the set of methods outside R to which objects created at S escape.

In order to compute the cdus that would cover interactions between R and I due to the statement S , we need to compute the new ape graph for the method M containing S as if regions R and I were integrated into a single region of focus. This requires computing the call subgraph of R rooted at M , and processing the ape graphs for each method in that call subgraph in reverse topological ordering. As each ape graph is built, if a call site to a method in I is encountered, the ape graph for the caller is incrementally updated to incorporate the ape graph for the method in I as part of the new integrated region.

The ability to incrementally update ape graphs in this manner is due to the compositional nature of the basic points-to escape graph. Specifically, the ape graphs for the original region of focus R were built by skipping over call sites to methods outside R . Now, as we integrate regions for integration testing, we can easily add the ape graphs for the skipped methods that are part of I to the already constructed

Algorithm: Object-based_Interactions(R,S,M)

Input: region of focus, R
statement creating objects of interest, S
method M containing S

Output: set of methods interacting with objects at S

Identify and reverse topsort subgraph G_M
of call graph rooted at M
foreach method N in top order in G_M **do**
 Construct ape graph ape_N for N

Let $escaped = \emptyset$
foreach inside node O of ape_M representing S
 $escaped.add(ape_M.escape(O))$
return $escaped$ // escaped objects created at S

Algorithm: Region-based_Interactions(R)

Input: region of focus, R
Output: methods interacting with objects created in R

Identify and reverse topsort call graph G_R
Let $escaped = \emptyset$
foreach method M in top order in G_R **do**
 Construct ape graph ape_M for M
 foreach inside node O of ape_M
 $escaped.add(ape_M.escape(O))$
return $escaped$

Figure 4. Identify Interactions Algorithms.

graphs of region R . The incremental update is performed according to Vivien and Rinard's description in [26]. Because callee ape graphs in I are merged into the callers' ape graphs in R where these call sites were previously skipped in the analysis, there is a propagating effect carried throughout the analysis of ape graphs until the root of the call graph is reached. Since we are only interested in cdus with respect to statement S , we need only perform the analysis until the method M is reached.

After the ape graphs are updated for region integration, cdus for objects created at statement S are constructed as described in the regional unit coverage algorithm. It should be noted that a new set of cdus is computed for S and the old set is discarded because many of the object relationships with respect to S could have changed.

For space reasons, we have not shown the algorithm for `All_Object_Integration_Coverage`. This algorithm parallels the `Object_Integration_Coverage` algorithm, with the difference that there is no specific subregion of the outside region to be integrated. The interacting region is everything outside of R , thus allowing us to calculate all object interactions that occur between R and the outside region.

For all of these algorithms, it is important to note that due to the compositional nature of the ape graphs, ape graphs can be constructed either on demand or cached and used as needed.

Algorithm: Regional_Unit_Coverage(R,S,M)

Input: region of focus, R
statement creating objects of interest, S
method M containing S
Output: set of cdus for objects created at S

Identify and reverse toposort subgraph G_M
of call graph rooted at M

foreach method N in top order in G_M **do**
 Construct ape graph ape_N for N

//calculate cdus by analyzing ape graph of M

foreach edge e in ape_M **do**
 if (source(e).equals(inside) of object at S)
 //construct cdus
 foreach store on e **do**
 find corresponding loads on e

Algorithm: Object_Integration_Coverage(R,S,M,I)

Input: region of focus, R
statement creating objects of interest, S
method M containing S
specified region of interaction by user, I
Output: cdus for region R interacting with I

Set $IM_{set} = \text{Object-based_Interactions}(R,S)$

Identify and reverse toposort subgraph G_M
of call graph rooted at M

foreach method N in top order in G_M **do**

foreach call site c in N **do**
 if (callee(N,c) $\in (IM_{set} \cap I)$)
 incremental-update(N,c)

//calculate cdus by analyzing ape graph of M

foreach edge e in ape_M **do**
 if (source(e).equals(inside) of object at S)
 foreach store on e **do**
 find corresponding loads on e

Figure 5. Integration Coverage Algorithms.

4. Regression Testing

In this section, we demonstrate how escape analysis could be useful when selectively retesting a portion of software that has been affected by a modification, instead of retesting all of the code by rerunning all tests in a test suite. Rothermel presents a description of the problems to be addressed in a selective retest process for regression testing [21]. The regression test selection problem is the problem of determining which test cases from the original test suite are needed to retest the modified code. A related problem is determining which test cases are obsolete and no longer valid due to code modifications. The problem of determining coverage involves identifying the portions of the modified software that require additional testing. A number of research groups have examined the problem of regression testing for object-oriented software [13, 28, 14, 20, 11, 10]. We believe that escape analysis information can provide valuable information for regression testing, particularly for selecting a subset of test cases from the original test suite

and in creating new test cases for testing the modified code.

We first describe the general situation where regression testing is needed and discuss how escape information could be quite useful. Then, we present an algorithm for creating additional cdu coverage and an algorithm for identifying obsolete cdus in response to particular code modifications.

Consider the situation when a fault occurs and a programmer fixes the source code. We perform regression testing in order to ensure that the modified region does not adversely affect the rest of the code. First, escape analysis could be performed with the set of modified methods specified as the region of focus, while the unchanged code serves as the outside region. The escape analysis results would provide information about all the occurrences of objects created in the modified region that interact with the outside region. This could help in selecting the test cases that need to be run in order to verify that the programmer's modifications are correct and do not adversely affect the outside region. Second, in addition to selecting the test cases, the escape analysis could also be used to provide coverage information, i.e., used to insure that interactions with the outside region are covered. Third, another valuable use of the escape information is determining previously computed cdus that are no longer valid, which in essence could be used to select the test cases associated with cdus that are obsolete.

The first algorithm shown in Figure 6 presents a solution to the problem of determining additional coverage needed when a new component A is added along with calls to that component from the region of focus R. The output of the algorithm is coverage for all the interactions between the region of focus R and the added region A, and region A itself. The algorithm begins by first determining the interactions between the region R and the newly added region A. A set of interacting methods is determined by calling the previously described Region-based_Interactions routine. A call subgraph for the region R is constructed, which may actually be a set of call graphs for the region, depending on how the methods in R interact. Then, for each method, we determine if a callee of any methods in R are in the newly specified region. If so, the incremental update of the graph as described previously is invoked to update the ape graphs of R. After all the updates to graphs have been performed, the new cdus are constructed. In order to construct the new cdus, we must process each ape graph in R and A.

The second algorithm presented in Figure 6 can be used to identify obsolete cdus upon adding a new component A. This algorithm could be invoked when a class implementation changes, but the interface remains unchanged. Therefore, the code of the region of focus R does not change, but needs to be retested to determine if the newly implemented instances of the modified class A adversely affect the code in R. The algorithm begins by first determining all of the interactions of the region R with the outside region. We an-

Algorithm: Regression_Test_Added_Region(R,A)

Input: region of focus, R
added region, A

Output: new cdus needed to cover R+A and A

Set $IM_{set} = \text{Region-based_Interactions}(R)$

Identify and reverse toposort set of call graphs for R

foreach method N in top order in G_R **do**

foreach call site c in N **do**

if (callee(N,c) $\in (IM_{set}) \cap A$)

 incremental-update(N, c)

//calculate cdus by analyzing ape graph in R + A

foreach ape graph $ape_a \in R + A$ **do**

foreach unmarked edge $e \in ape_a$ **do**

if(source(e).equals(inside))

foreach *store* on edge e **do**

 find corresponding *loads* on e

Algorithm: Delete_Obsolete_cdus(R,A)

Input: region of focus, R
added region, A

Output: set of obsolete cdus

Set $IM_{set} = \text{Region-based_Interactions}(R)$

foreach method M $\in R$

foreach call site in M

if($M_{callee} \in IM_{set} \cap A$)

Delete all cdus for callsite(M, M_{callee}) //obsolete

Figure 6. Regression testing algorithms.

analyze the methods that interact with the altered class implementation A. For each method invocation in R of a method in A, we delete all cdus that contain the line number of the call site pertaining to the invoked method in A. Note that no new ape graphs need to be created to determine obsolete cdus.

5. Empirical Results

In order to evaluate the effectiveness of escape analysis for software engineering applications, we have altered the FLEX compiler from MIT [19] to easily analyze arbitrary regions of Java code, as well as implemented the ape graph program representation by adapting the points-to escape graph implementation, and added instrumentation capability for automatically gathering metrics. In our experiments, we have defined the user code to be the region of focus R, and the Java library to be the outside region O. Table 1 shows some general characteristics of our Java benchmark programs and the number of interactions computed by different metrics. The benchmark characteristics include the number of lines of user code, the number of JVM instructions, the number of analyzed classes, and the number of analyzed methods. These numbers are reported separately for user and library sizes in order to show the differences in the region of focus R (user) and the outside region O (library).

Our goal in our experimental study was to gain some insight into the potential effectiveness and practicality of testing and retesting based on escaping objects. To better understand how escape analysis can be of use, we provide results illustrating how much information escape analysis provides about interactions between the region of focus and the outside region, relative to other common measures. We utilized a call graph construction algorithm similar to Age-sen's CPA [1], which is more precise than call graph construction algorithms like CHA and RTA [7, 2]. The *call graph* column in the table (column 10, CG) represents the number of caller/callee interactions between R and O based on the call graph edges only. These counts reflect the possibility of several method invocations at a single call site, (i.e., polymorphic call sites are counted once for each possible callee). The *parameter* column (column 11, param) illustrates the possible object interactions based on parameter counting from R to O. Only non-primitive type objects are counted, therefore it is possible that the number of reported object interactions is less than the number of caller/callee interactions. The *escape* column provides the number of object interactions based on escape analysis. The final column represents the average number of interactions per object based on escape analysis. The first three columns demonstrate that escape analysis provides more detailed information about how objects interact between regions of the program than a simple analysis over the call graph. By focusing the analysis on an object basis, we are able to provide a more fine detailed analysis of object interactions between regions. Although the number of interactions based on escape analysis seems quite large for some of the benchmark programs, the average number of interactions per object provides insight into how an object based analysis can be useful and scalable by analyzing a small number of objects at a time.

An important factor in determining the usefulness of escape analysis for software engineering applications is the time and space requirements for the analysis. A whole program escape analysis is costly, but the analysis of arbitrary regions of interest is significantly less expensive since the entire program representation for each method does not have to be constructed. By analyzing an arbitrary region of code, the points-to escape graph merge algorithm is not invoked on method calls to the outside region. Therefore, the analysis time and space is significantly reduced.

We also computed reduction in time and space for analyzing the entire program versus only an arbitrary region. On average, analyzing region R versus a whole program analysis (R + O) provided a 3.3 times savings, and on average, analyzing region R versus the whole program required 1.6 less space. From these results, we can conclude that program analysis and testing tools based on arbitrary regions of analysis are practical and can be beneficial in providing

Name	Problem Domain	# of lines	jvm instr		classes		methods		Number of Interactions			
			User	Lib	User	Lib	User	Lib	CG	Param	Escape	Ave
compress	text compression	910	2500	7070	17	90	50	301	876	1363	1444	1.54
db	database retrieval	1026	2516	11648	9	100	240	306	1146	1454	2836	1.64
jlex	scanner generator	7500	11000	7250	19	72	106	264	1050	557	7476	1.45
jess	expert system	9734	15200	13005	108	105	468	436	2272	1896	23757	2.06
jack	parser generator	7500	22633	17547	59	149	317	552	2501	2250	26157	1.15
jasmin	java assembler	7500	19405	19469	56	137	258	582	1181	309	7833	1.34
echo	debug web server	400	322	17205	3	142	14	526	83	41	360	1.55
jload	java installer	200	342	18520	1	152	3	611	91	43	307	1.71
log	message log	300	125	16608	2	131	3	498	37	23	79	1.53

Table 1. Program characteristics.

valuable information based on object interactions.

6. Conclusions and Future Work

The main contribution of this paper is the development of a novel testing and retesting approach that provides both flexibility and scalability in the testing and software maintenance process. By allowing arbitrary units of testing and analysis, and integration testing based on the interactions of particular objects, testers and software maintainers can focus their efforts more on particular object interactions, and save on testing resources.

References

- [1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proc. of ECOOP*, 1995.
- [2] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. of OOPSLA*, 1996.
- [3] H. G. Baker. Unify and conquer (garbage, updating, aliasing...) in functional languages. In *Proc. of the Conf. on Lisp and Functional Programming*, 1990.
- [4] B. Blanchet. Escape analysis for object oriented languages: Application to Java. In *Proc. of OOPSLA*, 1999.
- [5] J. Bogda and U. Hoelzle. Removing unnecessary synchronization in Java. In *Proc. of OOPSLA*, 1999.
- [6] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. of OOPSLA*, 1999.
- [7] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. of ECOOP*, 1995.
- [8] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proc. of POPL*, 1990.
- [9] B. Goldberg and Y. Park. Escape analysis on lists. In *Proc. of PLDI*, 1992.
- [10] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proc. of OOPSLA*, 2001.
- [11] P. Hsia, X. Li, D. Kung, C.-T. Hsu, L. Li, Y. Toyoshima, and C. Chen. A technique for the selective revalidation of oo software. *Soft. Maintenance: Research and Practice*, 9:217–233, 1997.
- [12] P. C. Jorgensen and C. Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–8, 1994.
- [13] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y.-S. Kim, and Y.-K. Song. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–86, 1995.
- [14] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. On regression testing of object-oriented programs. *The Journal of Systems and Software*, 32(1):21–40, 1996.
- [15] J. D. McGregor and T. D. Korson. Integrated object-oriented testing and development process. *Communications of the ACM*, 37(9):59–77, 1994.
- [16] A. Orso. *Integration Testing of Object-Oriented Software*. PhD thesis, Politecnico Di Milano, 1998.
- [17] J. Overbeck. *Integration Testing for Object-Oriented Software*. PhD thesis, Vienna University of Technology, 1994.
- [18] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Soft. Engineering*, 11(4):367–375, 1985.
- [19] M. Rinard. FLEX: The FLEX Group. January 2000. <<http://www.flex-compiler.lcs.mit.edu>>.
- [20] G. Rothermel, M. Harrold, and J. Dedhia. Regression test selection for C++. *Journal of Software Testing, Verification, and Reliability*, 10(6):77–109, 2000.
- [21] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proc. of ISSTA*, 1994.
- [22] G. Rothermel and M. J. Harrold. A coherent family of analyzable graph representations for object-oriented software. OSU-CISRC-11/96-TR60, The Ohio State University, 1996.
- [23] E. Ruf. Effective synchronization removal for Java. In *Proc. of PLDI*, 2000.
- [24] A. L. Souter and L. L. Pollock. OMEN: A strategy for testing object-oriented software. In *Proc. of ISSTA*, 2000.
- [25] A. L. Souter and L. L. Pollock. Contextual def-use associations for object aggregation. In *Proc. of PASTE*, 2001.
- [26] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proc. of PLDI*, 2001.
- [27] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. of OOPSLA*, 1999.
- [28] L. J. White and K. Abdullah. A firewall approach for regression testing of object-oriented software. In *Software Quality Week*, 1997.