OMEN: A Strategy for Testing Object-Oriented Software *

Amie L. Souter
Computer and Information Sciences
University of Delaware
Newark, DE 19716
souter@cis.udel.edu

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

ABSTRACT

This paper presents a strategy for structural testing of object-oriented software systems with possibly unknown clients and unknown information about invoked methods. By exploiting the combined points-to and escape analysis developed for compiler optimization, our testing paradigm does not require a whole program representation to be in memory simultaneously for testing analysis. Potential effects from outside the component under test are easily identified and reported to the tester. As client and server methods become known, the graph representation of object relationships is easily extended, allowing the computation of test tuples to be performed in a demand-driven manner, without requiring unnecessary computation of test tuples based on predictions of potential clients.

1. INTRODUCTION

The effectiveness of control flow and data flow-based testing techniques is unclear in the object-oriented programming domain where object-oriented design principles result in programs with a structure that differs significantly from the imperative programs originally targeted by these methods. The novel characteristics of object-oriented software are primarily due to classes, inheritance, polymorphism and dynamic binding. Applications consist of a set of classes (with a main program or main class), which contain calls to methods within these classes as well as calls to methods in classes written by others. A programmer may design and implement just a single class or library as their designated programming goal, without a specific targeted application which will use the class. The context of the class is often unknown. Moreover, the order in which methods of the class will be called from client applications is unknown.

With the evolution of object-oriented software into component-based software comes an additional set of characteristics affecting the testing process[27]. A component may not necessarily be a class or a method, but any grouping of methods. Objects may leave one component and become visible to another component, through parameter passing and return statements. An application can have a changing composition as components are added, replaced, updated, and deleted. Incomplete programs and programs with portions unknown to the analysis need to be addressed. Some components may be in binary only and dynamically loaded.

In this paper, we propose a new paradigm for structural testing of object-oriented software. The premise behind our approach is that object manipulation is a natural testing basis for exposing the possible behaviors of object-oriented software, and that a testing tool should be capable of aiding a programmer in testing object manipulations in a component-based programming environment. Our approach was inspired by both the characteristics of component-based software and the results from our empirical study of the characteristics of a set of large Java applications, which we reported in a previous paper[24]. We found that object-oriented design commonly results in many methods, each with a small number of statements and simple intraprocedural control flow. The static branch count within a method is often zero, and on average between 0 and 3. This suggests that control flow-based testing may not be the most effective for revealing the behaviors of the program. Often a small percentage of methods defined in a server class are actually used by a given client class. Only approximately half of the methods used by the client actually change the state of the object being used within the client. Very few manipulations of primitive-type variables typically targeted by data flow testing occur in the programs. Instead, computation is achieved primarily through manipulation of instance variables of objects via method calls. Polymorphism and loads and copies of object references create aliasing relationships that need to be considered in the identification of object manipulations for testing.

We begin by illustrating the testing coverage provided by existing techniques applicable to object-oriented software. We then introduce the object manipulation-based testing approach by first identifying and classifying the set of possible object manipulations followed by a description of our approach to achieving object manipulation-based testing. The key insight is that some simple extensions to the points-to escape graph introduced by Whaley and Rinard[29] for compiler optimization enable the computation of related ob-

*This work was supported in part by NSF EIA-9806525 and NSF EIA-9870370.

ject manipulations which can serve as a basis for object manipulation-based testing, without requiring a whole program representation in memory simultaneously.

Our extended points-to escape graph representation, which we call the *ape* (Annotated Points-to Escape) graph[1], contains adequate information to identify potential manipulations to the same object in the presence of polymorphism, aliasing, and inheritance, in addition to knowledge of which manipulations are captured within the software component being tested, which objects may be potentially manipulated from outside the component, and the different ways in which the object is accessible from outside. With this representation of object manipulations, we can identify object creation and potential write's and read's to the same object for testing, similar to data flow testing of primitive-type variables, in a straightforward way. Furthermore, we are able to give various kinds of feedback to the tester about testing objects with respect to the interactions with other unknown components.

2. MOTIVATING EXAMPLE

Consider data flow testing of the object-oriented program in Figure 1 based on covering def-use associations. If we apply several known def-use analysis techniques to the code, we obtain the def-use associations shown in Table 1. Each approach extends the previous approaches to report a superset of the def-use associations of the approaches in all previous rows of the table. Thus, the indicated def-use associations for each row are def-use pairs identified in addition to all those in previous rows. The notation methodname-list(variable name, def statement, use statement) represents the def-use association corresponding to the sequence of method calls in the methodname list.

By using traditional def-use analysis for primitive types only, we obtain only one def-use pair, `main(i,19,19)`. This def-use pair is formed from the definition of i on line 19 and its use of i also on line 19. Obviously, this does not give us much information about the program. Next, using an intra-method def-use analysis applied to any variable, not just primitive types in the method, we obtain several def-use pairs. Intra-method def-use pairs have the same meaning as they do in a procedural language; a definition of a variable and a subsequent use are both located in the same method. An example of such a def-use pair is `pop(t, 8, 14)`, where t is defined on line 8 of the method `pop` and then used on line 14. The resulting set of pairs is limited in its testing coverage, because it does not test the interactions between different methods.

Next, we consider inter-method def-use pairs. Inter-method def-use pairs have the same meaning as interprocedural def-use pairs in a procedural language. However, inter-method def-use pairs result from the calling relations between methods of a class disregarding the unknown sequence of calls after instantiation[14]. In this example, an inter-method approach does not give us any additional pairs due to the fact that the only inter-method calls invoke methods of other classes. Intra-class def-use pairs are created by sequences of

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

```
   class Stack{                              //Basic Operations
     Node top;
1      public Stack(){ top = null; }         //store: this.top
2      public void push(Object e) {
3         if (top == null)                   //load: r1 = this.top
4             top = new Node(e, null);       //object creation,
5         else                               //store: this.top
6             top = top.insert(e);  }        //call, store:
                                             //this.top
7      public Object pop(){
8         Object t = null;
9         if(isempty())                      //call
10           System.out.println("ERROR: nothing to pop");
11        else{
12            t= top.get();                  //call
13            top = top.remove();  }         //call, store:
                                             //this.top
14        return t; }
15     public boolean isempty() {
16           return top == null; }  }        //load:
                                             //r2 = this.top
    class StackClient{
17     public static void main(String args[]){
18        Stack s = new Stack();             //object creation
19        for(int i = 0; i < 10; i++)
20            s.push(new Integer(i*2));      //call
21        while(!(s.isempty())){             //call
22            Object x = s.pop();            //call
23            System.out.println(x); } } }
   class Node{
     Object data;
     Node   next;
24     Node(Object e, Node n) {
25       data = e;                           //store: this.data
26       next = n;  }                        //store: this.next
27    Object get() { return data;}           //load:
                                             //r2 = this.data
28     Node insert(Object e){
29       Node temp = new Node(e, this);      //object creation
30       return temp;     }
31    Node remove(){
32       Node e = this;                      //copy
33       e = e.next;                         //load: e = e.next
34       return e;           }  }
```

Figure 1: Object-oriented stack implementation

method invocations that arise if the class was instantiated. An instantiated class can call methods in any order. Using an approach based on intra-class pairs, as defined in [14], we obtain several additional def-use pairs. For example, from two successive calls to `push`, we obtain the def-use pair `(top, 4, 3)`, where we define *top* in the first call on line 4, and then use it in the second call to `push` on line 3. This approach obtains several additional def-use pairs, but again it does not exercise the behavior of the object-oriented nature of the code.

The last two approaches begin to use objects as the basis for def-use pairs. In order to obtain these def-use pairs, a technique needs to incorporate the following information into its analysis:

- treat the receiver object as a parameter to method calls
- associate accesses to instance variables with fields of objects passed as parameters
- view definitions of the field of an object passed into a method as a definition of the object itself

Research in optimization of object-oriented programs has commonly treated the receiver as the first parameter [29]. Associating field names with objects is similar to that of fields being associated with structs in C, and researchers have made this association for C programs in conjunction with pointer analysis[31]. Finally, viewing the definition of an object as any definition of its field variables was introduced in [24].

The object-state entry of the table shows the additional def-use pairs obtained by incorporating this additional information into the analysis. For example, we obtain the def-use pair `(s, 18, 20)` in `main`. The use of *s* on line 20 of `main` is identified by an analysis that identifies the use of instance variable *top* in the method `push`, and the fact that instance variable *top* is associated with the object *s*. The pair `(s, 18, 21)` is another example of a def-use pair found using this method.

The inter-class approach extends the intra-class and object-state techniques by analyzing a single class's manipulation of objects of other classes [24]. For example, the def-use pair `push-pop-[remove:Node](top, 6, 13)` corresponds to the definition of *top* on line 6. A subsequent call to `pop` uses *top* at line 13, by following the call to remove, where *top* is used on line 32.

Each of these approaches falls short in exploiting the association between objects and their fields or instance variables. The last two approaches begin to form this association, but are not complete. None of the above approaches uses points-to information in its analysis, to consider the cases where several references refer to the same object. There is also no means to provide the tester with any information when part of the code is missing in the case of a library or third party code. All of the code must be analyzed for the analysis to be complete.

Approach	Example Additional Def-Use Pairs
primitive types	main (i,19,19)
intra-method	pop (t,8,14), pop(t,12,14), insert (temp,29,30) remove(e,33,34)
inter-method	no additional def-use pairs
intra-class	push-push(top,4,3), push-push(top,6,3) pop-pop(top,13,12), pop-push(top,13,3)
object-state	main(s,18,20), main(s,18,21)
inter-class	push-pop-[remove:Node](top, 6, 13(32)) push-push-[insert:Node](top,6, 6(29))

Table 1: Def-use pairs from existing analyses

3. THE OMEN APPROACH

Object-oriented programming focuses on the data to be manipulated rather than the procedures that do the manipulating. An object-oriented program achieves its goals by creating objects of specific classes. The state of an object is encapsulated as a copy of all of the fields of data that are defined in the corresponding class definition. Actions are performed on an object by invoking methods defined in the class definition, often called sending a message to the object. A method invocation can modify and/or read the data stored in the particular object.

Object-related Statements	Object Manipulations
copy $r_1 = r_2$	read of reference r_2 write to reference r_1
load $r_1 = r_2.f$	read of reference r_2 read of field $r_2.f$ write to reference r_1
store $r_1.f = r_2$	read of reference r_2 read of reference r_1 write to field $r_1.f$
global load r = cl.f	read of class variable f write to reference r
global store cl.f = r	read of reference r write class variable cl.f
return r	read of reference r
object creation r = new Object(....)	create a new object write to reference r MOD and USE
method invocation $r = r_0$.methodname(r_1,..., r_n)	write to reference r read of references r_0-r_n MOD and USE of r_0's fields

Table 2: Basic object manipulations

In order to better understand the possible behaviors of an object-oriented program in terms of object manipulations, we identify the most elemental object manipulation as either a read or write action. The actions that a particular statement or method performs on an object can be decomposed into a sequence of these elemental actions. Specifically, we identify the following set of basic read and write actions with respect to objects and their references:

- read the value in a reference to an object
- write a value to an object, usually making the reference point to a different object
- read the value in a field of an object
- write a value in a field of an object
- create a new object
- delete an object (e.g., C++)
- pass a reference to an object as a parameter
- return a reference to an object from a method

There are also basic read and write actions on class instance variables. These basic object and static class variable manipulations manifest themselves in programs through the use of statements of the forms given below. The syntax $r.f$ represents an access to the field f of the object referenced by r, and $cl.f$ denotes the static class variable f of class cl. Due to aliasing and polymorphism, we may have a set of objects potentially referenced by each reference, but for these descriptions, we use the singular form. However, our analysis addresses the potential for a set of objects being referenced.

- A *copy* statement, $r_1 = r_2$, sets the reference r_1, to point to the same object referenced by r_2; r_1 no longer

points to the object it previously referenced (unless it was also referenced by r_2).

- A *load* statement, $r_1 = r_2.f$, sets the reference r_1 to point to the object referenced by the field f of the object referenced by r_2.
- A *store* statement, $r_1.f = r_2$, sets the field f of the object referenced by r_1 to point to the object referenced by r_2.
- A *global load*, $r = cl.f$, sets r to point to the same object referenced by the class variable f of cl.
- A *global store*, $cl.f = r$, sets the class variable f of cl to point to the same object referenced by r.
- A *return* statement, *return* r, indicates that the method in which the return statement is located returns the object referenced by r.
- An *object creation*: For r = *new Object(...)*, an object of the class *Object* is created, and r is set to point to the new object. Parameters are handled analogous to other method invocations.
- A *method invocation*, $r = r_0.methodname(r_1, r_2, ...r_n)$ results in invoking the method called *methodname* of the object referenced by r_0, passing object references $r_1, r_2, ...\ r_n$ as parameters, and returning an object which is then referenced by r after the call completes. The reference r_0 is implicitly passed as a parameter to the method.

Table 2 depicts the elemental object manipulations performed by each of these statements. We assume that the program has been preprocessed such that all statements that perform object manipulations have been expressed in the form of these basic statements.

In this paper, we present an approach to structural testing of object-oriented software that is based on providing coverage in terms of the elemental read and write actions. We call our approach the *OMEN* approach, because it is based on covering basic *O*bject *M*anipulations in addition to using *E*scape i*N*formation to provide helpful feedback to the tester in an interactive testing tool environment.[2] While our eventual goal is to develop a set of testing criteria under the *OMEN* approach, this paper focuses on demonstrating one specific testing technique using this approach.

We extrapolate the concept of data flow testing to the testing of elemental object manipulations by defining a *(write, read)* association of a given object's state, extending this association to include object creation points, and developing an algorithm for computing these test tuples. Recall that an object's state is captured by its local copy of the fields defined in the corresponding class definition. From Table 2, we can see that the statement that reads an object field is the load statement, while the store statement writes to an object field. To ensure that we do not miss any viable (write, read) pairs, we assume that a given load/store statement may read/write the field of any object to which the reference is potentially referencing at that program point.

For a given load statement l: $r_1 = r_2.f$, the set of (write, read) pairs associated with the read at l, WR-pairs(l), is defined as follows:

> Let $pro = \{o \mid o$ is an object potentially referenced by $r_2\}$. WR-pairs(l) is defined to be the set of pairs of the form:
> *(w, l)* where $w \in \{s \mid s$ is a store r_3.f $= r_4$, such that r_3 potentially references an object o in pro, and s reaches l along some definition-free path for the field f of $o\}$.

Because objects are instantiated at run-time through executable statements, we extend (write, read) pairs to triples of the form (write, read, object creation) to reflect the fact that a test case should cover the creation of the object before any write's or read's to that object. In the remainder of this paper, we present our method for computing these test tuples, determining when the tester needs to be aware of outside influences on the test results for particular objects, and when the tester may need to provide a driver to enable testing of particular objects.

4. PROGRAM REPRESENTATION

Each method's control flow is represented by a control flow graph(CFG)[28]. The calling relationships between methods are represented by a call graph, which could be constructed by any conservative call graph construction method[11, 2]. The relationships between objects are represented by an *ape* graph, our extension to the points-to escape graph representation developed by Whaley and Rinard[29].

4.1 The Points-to Escape Graph

The points-to escape graph representation combines points-to information about objects with information about which object creations and references occur within the current analysis region versus outside this program region[29]. The current analysis region is the region of the program on which program analysis is being performed, while the regions outside the current analysis region include routines either called by or calling methods in the current analysis region, in which the code is unavailable to the program analysis. The points-to information 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. Whaley and Rinard developed this representation to enable an optimization analysis of Java programs that determines which synchronization operations are unnecessary and could be eliminated, as well as for reducing the number of objects that are unnecessarily allocated on the heap when they could be allocated on the stack. Their analysis is able to analyze incomplete program units as well as arbitrary parts of the program, while providing complete information about objects that do not escape the analyzed region. Their analysis analyzes each method once to produce a single parameterized analysis result, allows for analyzing

[2]In addition, we view the test cases and the results of executing the test cases as an *omen* to predicting the behaviors of the executing program in the production environment.

a method independent of its callers and unanalyzed callees, and then combines analysis results from multiple methods through interprocedural analysis, and increases the precision of the results as invoked methods are analyzed.

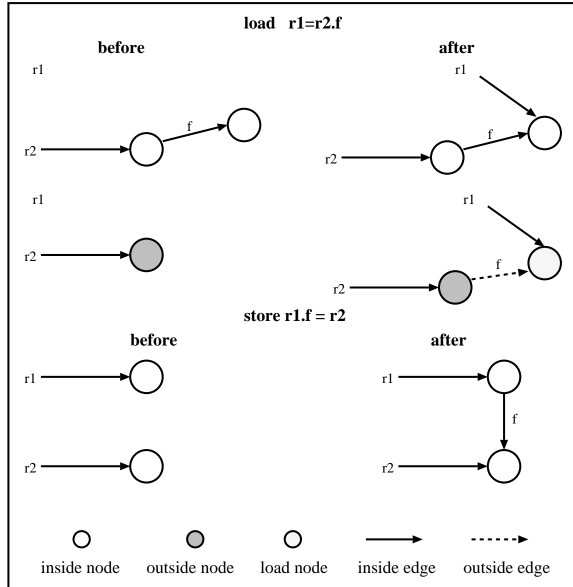


Figure 2: Points-to graph changes for load/store

In the 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 set of nodes in the points-to escape graph. Each *inside node* represents an object creation site for objects created and reached by references created inside the current analysis region of the program. A single inside node representing an object creation site represents all objects created at that site. A *class node* represents the statically allocated object with fields for the static class variables for the corresponding class. *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, namely, parameter nodes, load nodes, and return nodes. There is one *parameter node* for each formal parameter; the 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. 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. Finally, there is one *return node* for each method invocation site in the program to represent the return values of the method invocation of unanalyzed methods.

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 currently analyzed 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.

The distinction between inside and outside nodes is important because they are used to characterize nodes as either captured or escaped. A *captured* node corresponds to the fact that the object it represents has 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, an *escaped* node represents the fact that the object escapes to unanalyzed portions of the program. An object can escape in several ways. A reference to the object can be passed as a parameter to the current method, a reference to the object was written into a static class variable, a reference was passed as a parameter to an invoked method and there is no information about the invoked method, or the object is returned as the return value of the current method.

For a given method, an initial points-to escape graph is constructed to represent the parameters and class objects on entry to the method. The points-to escape graph is refined by repeatedly processing the object-related statements in the CFG of a method until a fixed point is reached. At each statement, points-to escape graphs representing the predecessors of the statement in the CFG are first merged into a single graph. The new points-to escape graph in effect at the program point immediately after the statement is constructed by applying the statement's transfer function to the merged graph that was in effect immediately before the statement. For example, in the analysis of a copy or load statement, the edges between a reference and the object which it references before the statement are deleted (or killed) when the reference points to a new object after the statement.

Figure 2 illustrates the changes in a points-to escape graph during analysis of load and store statements. The effect of a load $r_1 = r_2.f$ is represented in the graph in two different ways depending on what r_2 references before the load statement. In the first case, where r_2 points to no escaped nodes, the set of nodes, S, accessible via inside edges from $r_2.f$ is computed, the set of nodes that r_1 previously referenced is killed, and then inside edges from r_1 to all the nodes in S are generated. The second case occurs when r_2 references escaped nodes. In this case, the value stored in $r_2.f$ could be a reference created outside the current analysis region. Therefore, other parts of the program can reference and change the value of the field f. We have no knowledge of the objects possibly referenced by the field f. To model this case, the same process as above occurs, killing all edges from r_1 and generating inside edges to all inside nodes, but for all escaped nodes that r_2 references, outside edges are generated from the escaped nodes to the newly generated load node and labeled with the field name f. The execution of a store statement, $r_1.f = r_2$, results in r_1's f field pointing to the object that r_2 references. The graph represents this statement by finding the set of nodes that r_1 references, and then generating inside edges from all of these nodes to the nodes that r_2 references.

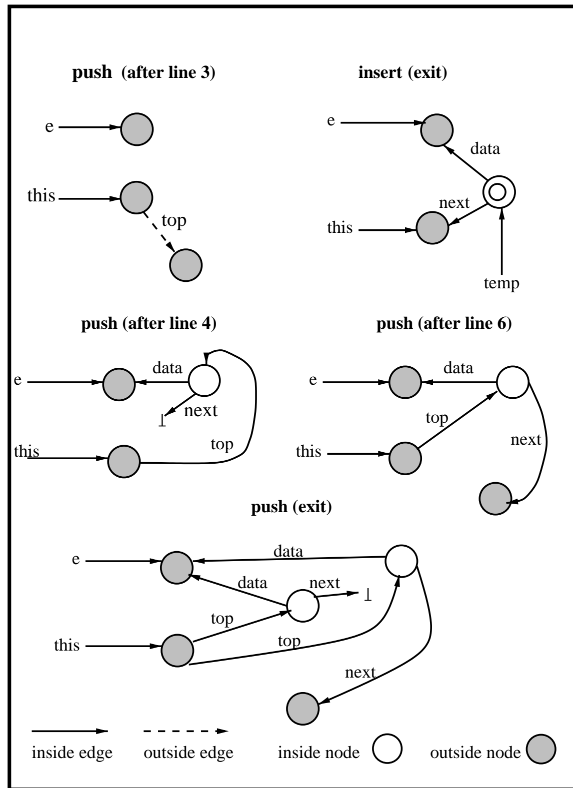


Figure 3: Points-to escape graph example

Interprocedural analysis is achieved through merging the parameterized points-to escape graphs of the potentially invoked methods at a 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. Nonrecursive programs can be processed in a reverse topological sort order, while recursive programs will involve fixed-point iterative analysis within each strongly connected component of the conservative call graph. For call sites to unanalyzed methods, the parameters and return value are marked as escaped within the caller's graph.

Figure 3 shows the computed points-to escape graphs for various points in the code given in Figure 1. The points-to escape graphs are shown for the end of the `insert` and `push` methods, as well as the graphs for several intervening program points within the method `push`. The first graph (`push` after line 3) shows the points-to escape graph for `push` prior to the conditional statement. The parameters *e* and *this* both point to outside nodes representing the fact that the object to which the parameters point were created outside the current analysis region, the `push` method. An outside edge for *top* points to a load node, indicating that the reference in statement 3 reads a reference created outside `push`. The graph for `insert`'s exit point has a similar structure for the parameters *e* and *this*. The reference *temp* points to an inside object (created inside the current analysis region, `insert`), whose *data* field points to *e* and *next* field which points to *this*. These edges are created through the call to the `Node` constructor. The inside node referenced by *temp* is also the return value of the `insert` method.

The graph for `push` (after line 4) illustrates the points-to graph that corresponds to the code following the true branch. The reference *top* now points to a new inside node created by the call to the `Node` constructor, with *data* field pointing to the parameter *e* and *next* field pointing to null. `Push` (after line 6) corresponds to the points-to graph representing the false branch of the `push` method. The reference *top* points to an inside node that corresponds to the return value of the method `insert`. Through interprocedural analysis when building this graph, the mapping of nodes from `insert` to `push` results in the *data* field of *top* pointing to the parameter *e* and the *next* field pointing to the old *top*, which is represented by the outside node with no label. Finally, the points-to escape graphs are merged together at the join of the if statement to result in the graph illustrating the final result at the exit of the method `push`. The fact that `insert` could be a virtual call site is handled through the use of a conservative call graph. Therefore, a set of nodes are mapped from `insert` to `push` for the potentially invoked `insert` methods.

4.2 The APE Graph

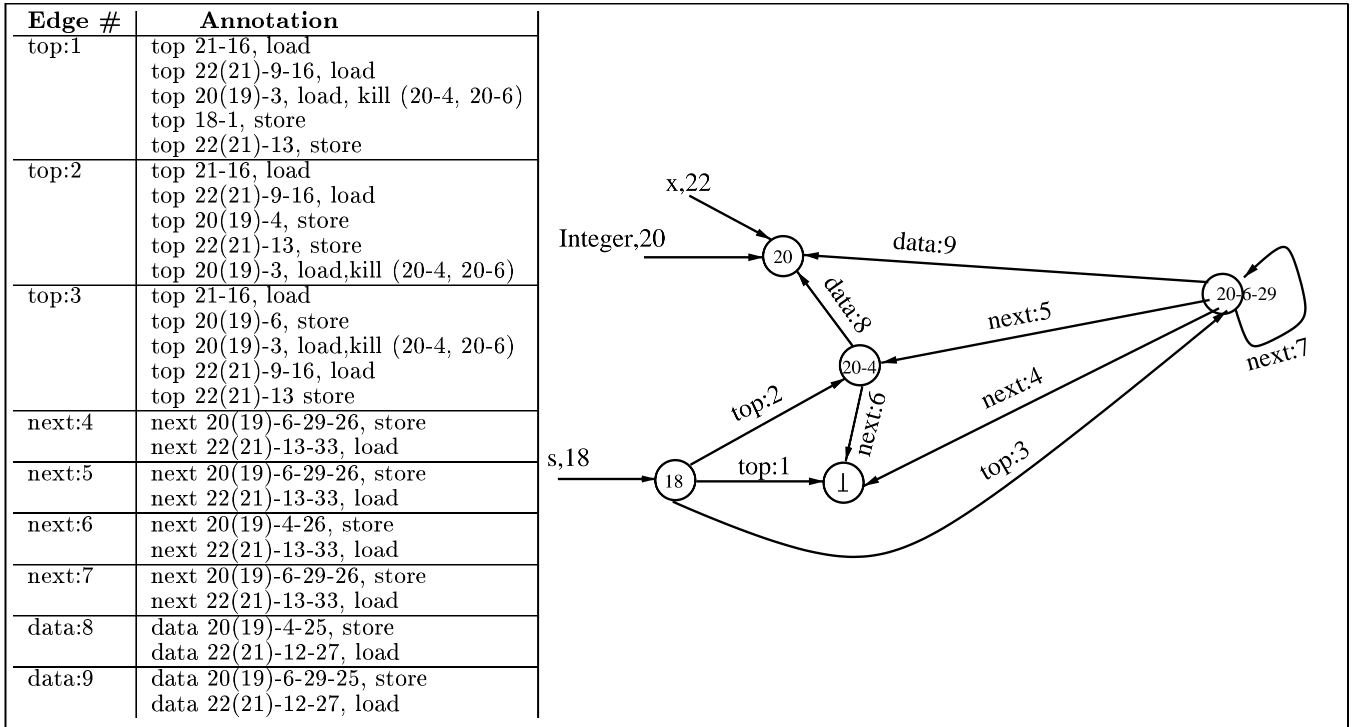
While Whaley and Rinard[29] compute the points-to graph at each program point during their analysis, the points-to escape graph for a given method represents the points-to and escape information with respect to the method exit. For our application of the graph, we need to know points-to escape information at each object-related statement. Our analysis needs to be able to answer questions such as: *What could be referencing a given object at a particular program point, such as at a given load statement? Similarly, what objects could be read at this point in the program? Where are all the possible reaching write's to this object prior to this program point?* However, we want to avoid the storage requirements for a points-to graph representation at every program point during testing analysis.

In order to avoid saving separate points-to graphs for each object-related statement, we have made the following modifications to the points-to escape graph produced by Whaley and Rinard[29] for each method exit, to enable pointwise information to be retained in a single representation for each method.

Edges are not deleted upon kills. Each edge is annotated with information about the loads and stores associated with the edge's reference. From the construction of the inside and outside edges, we know that an inside edge labeled with a field is only created by store statements. However, an inside edge can also be labeled with multiple load and store annotations, since an edge from node n_i to node n_j represents all references from n_i to n_j, and there could be multiple loads and stores of a reference from n_i to n_j. Outside edges labeled by fields are only created by load nodes.

For each load/store of the reference represented by a particular edge e, we logically maintain:

- a sequence of statement numbers, $(s_1, s_2, ..., s_n)$, where s_n is the unique statement number of the load/store statement; $s_1, s_2, ...s_{n-1}$ contains the statement numbers of the call sites where this edge was merged into



Edge #	Annotation
top:1	top 21-16, load top 22(21)-9-16, load top 20(19)-3, load, kill (20-4, 20-6) top 18-1, store top 22(21)-13, store
top:2	top 21-16, load top 22(21)-9-16, load top 20(19)-4, store top 22(21)-13, store top 20(19)-3, load,kill (20-4, 20-6)
top:3	top 21-16, load top 20(19)-6, store top 20(19)-3, load,kill (20-4, 20-6) top 22(21)-9-16, load top 22(21)-13 store
next:4	next 20(19)-6-29-26, store next 22(21)-13-33, load
next:5	next 20(19)-6-29-26, store next 22(21)-13-33, load
next:6	next 20(19)-4-26, store next 22(21)-13-33, load
next:7	next 20(19)-6-29-26, store next 22(21)-13-33, load
data:8	data 20(19)-4-25, store data 22(21)-12-27, load
data:9	data 20(19)-6-29-25, store data 22(21)-12-27, load

Figure 4: Ape graph for exit of main

the caller's ape graph during interprocedural analysis performed during construction of the current analysis method's ape graph. Statement s_1 is the statement number of the call site within the current analysis method which eventually leads to the load/ store statement.

- a corresponding sequence of statement numbers, (evs_1, $evs_2, ...evs_n$), where each evs_i is the unique number of the earliest statement at which the statement s_i could have an effect on other statements. We call this the earliest visible statement for s_i, evs_i. The earliest visible statement $evs_i = s_i$ when the statement s_i is not inside a loop; otherwise $evs_i =$ the statement number of the header of the outermost loop containing s_i.
- a corresponding sequence of sets, $kill_{s_i}$, of statement numbers corresponding to the statements where the reference at s_i would be killed and removed by the points-to escape graph construction.

Statement number sequences also label nodes to indicate the statement that created the node and calls leading to that statement from the current analysis method.

The statement numbers are needed for our analysis in determining the order of manipulations to objects. The earliest visible statement numbers are needed to correctly determine the first visible point of references created within loops. The kill information is needed to gain more precise information about points-to relations that hold at a given point in our analysis.

Sequence of statement numbers, earliest visible statements, and kill sets are used to correctly compute write-read tuples that cross over methods. The statement number of the call site within the current method is used for statement order information by the algorithm's processing within the current method when a load/store statement occurs in an invoked method. The statement number of the actual load/store statement is used to indicate the final testing tuple.

The sequence for a given load/store annotation 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.

The points-to escape graph merge algorithm is also extended to mark each callee's ape graph edge in the callee's ape graph representation that is mapped into a caller's ape graph. After a given method's graph has been mapped to possibly several caller's ape graphs, the ape graph for the method itself will have all edges marked which have been mapped to any caller's graph. All unmarked edges are edges not reachable by callers. These marks are used in the test tuple construction algorithm to avoid creating duplicate test tuples and also to identify loads that may have reaching stores outside the component under test.

Figure 4 shows the ape graph for the exit of `main`, assuming that we have knowledge of all invoked methods from `main`, including methods in the `Node` class. For space reasons, we only show evs_i in parenthesis when evs_i is different from s_i.

As an example, the edge labeled top:1 is annotated by two stores and three loads. The load 21-16 represents the load to *top* in `isempty` called from the call site at statement 21 in `main`. The merge of the `isempty` graph with the graph for `main` at the call site at line 21 concatenated the 21 to the statement sequence on the annotation for the load. Figure 6 shows the ape graph for the exit of `main` when the `Node` class is unknown to the analysis of `main`.

5. TEST TUPLE CONSTRUCTION

Test Tuple Construction Algorithm: Compute testing tuples for an object-oriented component.

```
Input: set of call graphs for component under test, CUT;
one ape graph per method in CUT;
Output: set of testing tuples for CUT;
feedback on potential influences from outside CUT;

Let T be a topological ordering of the nodes in the call graphs
for CUT, starting at the roots of each call graph;
foreach node n in T do
   Let AG_m = ape graph for method m represented by node n;
   /* create tuples from stores in m or stores reachable from m */
   foreach unmarked edge e in AG_m labeled by a STORE do
      foreach STORE s labeling e do
         Let cs_s = unique statement number for s in AG_m;
         Let evs_s = earliest visible statement for s in AG_m;
         foreach LOAD l annotation on e do
            Let cs_l = unique statement number for l in AG_m;
            if evs_s < cs_l then
               Create a tuple of the form (store,load)
               where store = (cs_s-remaining statement sequence for s),
               and load = (cs_l-remaining statement sequence for l);
               /* find object creation site for this store-load pair */
               Let sn = source node of e;
               Let cs_sn = unique statement number for sn in AG_m;
               if sn is an inside node then
                  Replace tuple (store,load) by (store,load,cs_sn);
               else /* sn is an outside node */
                  if n is a root in a call graph for CUT then
                     Feedback(object for (store,load) is potentially
                              created outside CUT);
                  else /* n is an interior node in call graph */
                  if sn is not a parameter node then
                     Feedback(object for (store,load)is potentially
                              created outside CUT);
            Let tn = target node of e;
            if sn not escaped and tn is escaped then
              Feedback(value loaded in (store,load,cs_sn) is po-
                       tentially changed by method outside CUT,
                       but l is indeed referencing object created at
                       cs_sn);
         endfor
      endfor
   endfor
   /* examine loads in m or reachable from m */
   foreach unmarked edge e in AG_m labeled only by LOAD do
      Let tn = target node of e;
      if tn is a load node in AG_m then
         Feedback(object for (store,load) is potentially created
                  outside CUT);
         Feedback(load at statement cs_l in method m has potentially
                  reaching references from outside CUT);
   endfor
endfor
```

Figure 5: Test tuple construction algorithm

The test tuple construction algorithm in Figure 5 computes a set of testing tuples for the component under test (CUT), based on object manipulations. Starting at the root of each call graph of CUT and proceeding in topological order, the method for each call graph node is processed once, by analyzing the node's ape graph. This processing order avoids creating duplicate tuples potentially identified due to subgraphs of invoked methods also appearing in a caller's ape graph. As a particular ape graph is analyzed, only unmarked edges (those not already processed in a caller's graph) are processed.

The algorithm processes each edge in a method's ape graph. For each annotation on an ape graph edge representing a store, the associated loads potentially occurring after the store are identified, and a (store,load) tuple is created. The annotations reflect the results of the flow sensitive points-to escape analysis used to build the ape graph. Thus, the *evs* and *cs* statement numbers on these annotations are adequate to identify the reachable loads from a particular store. The object creation site associated with the (store,load) tuple is determined by the source node of the edge being analyzed. If the source node is an inside node, then the source node is the object creation site and the node number of the source node is used to complete the tuple for the (store,load) tuple. If the source node is an outside node, then the object is not created inside CUT, and feedback is given depending on the kind of the source node and whether it is interior or root of the call graph. Additionally, feedback is given when the target node of the ape graph edge being analyzed is escaped from CUT. In this algorithm, we do not utilize the kill information; however, it is used in algorithms we are currently developing, thus we included it in our graph description.

The algorithm also provides feedback for load nodes when a corresponding store is not present in CUT. This is represented by an ape graph edge that is labeled only with load annotations, and no store annotations. The feedback provides the tester with information about the fact that an object creation site could have potentially occurred outside CUT, as well as the possibility that the load in CUT has potentially reaching references from outside CUT.

5.1 Examples

5.1.1 *Complete Component*

Table 3 shows the set of tuples calculated using our algorithm and the ape graph illustrated in Figure 4. For space reasons, we only show the tuples when processing main. The first column indicates the ape graph edge which creates the testing tuple. The second column shows the tuples in the form object-name(store, load, object creation site). For example, the edges top:1, top:2, and top:3 create tuples due to the objects created at line 18 in the code in Figure 1. The tuple (<20-4>,<21-16>,18) created from edge top:2 corresponds to an object *s* created on line 18, which has a value stored at line 4, through the call site at line 20, and value loaded at line 16 through the call site at line 21. There is no feedback provided for this example because the component is a complete program.

5.1.2 *Component with Unknown Subcomponent*

The second example with its ape graph shown in Figure 6, does *not* have knowledge of the internals of `Node` class. This ape graph differs from the previous one due to the outside

Edge #	StackClient Tuples
top:1	top(<18-1>, <20-3>, 18)
top:1	top(<18-1>, <22-9-16>, 18)
top:1	top(<18-1>, <21-16>, 18)
top:1,2,3	top(<22-13>, <22-9-16>, 18)
top:2	top(<20-4>, <21-16>, 18)
top:2	top(<20-4>, <20-3>, 18)
top:2	top(<20-4>, <22-9-16>, 18)
top:3	top(<20-6>, <20-3>, 18)
top:3	top(<20-6>, <22-9-16>, 18)
top:3	top(<20-6>, <21-16>, 18)
top:1,2,3	top(<22-13>, <21-16>, 18)
next:4,5,7	next(<20-6-29-26>, <22-13-33>, 20-6-29)
next:6	next(<20-4-26>, <22-13-33>, 20-4)
data:8	data(<20-4-25>, <22-12-27>, 20-4)
data:9	data(<20-6-29-25>, <22-12-27>, 20-6-29)

Table 3: Computed tuples for complete program

nodes needed for the unknown return value at the call sites of methods in the `Node` class. The edge annotations are similar, except that the annotation details due to the `Node` class are unknown in this example. Also, this example has an additional edge, top:4, which the first example does not have because during the merge of the graphs at call sites in the first example, the annotations are attached to existing edges created from the `Node` class methods, which are not available in this example. Each tuple in this example will have associated feedback to the tester. The feedback provided corresponds to the line in the algorithm, where it is found that the source node is an inside node and the target node is an outside node. The feedback given is the fact that the value loaded at a particular point may be changed in a method outside CUT.

The algorithm would also work correctly if the `main` program were missing and we only had the `Node` class and/or `Stack` class. The feedback would inform the tester that the object creation site for the store/load pairs occur outside CUT. To thoroughly test the component, a driver would be needed that created the object of interest.

5.2 Space/Time Cost Analysis

We maintain one ape graph for each method in CUT. The worst case for an ape graph is to have it fully connected, that is, each object has a reference to every other object. In this case, the size of an ape graph would be n^2 for n object creation sites in a complete program. In a component with unknown subcomponents or callers, n would include both object creation sites inside CUT and load nodes representing objects potentially created outside CUT.

The testing tuple construction algorithm takes one pass over the call graphs representing CUT. Because of the way the ape graph is constructed interprocedurally, it is adequate to take one pass through the nodes of a cycle in the call graph. For each node in the call graph, it processes each unmarked edge of the ape graph for that method exactly once. In the worst case, the number of edges processed by the algorithm is the total number of edges in all ape graphs of CUT. However, due to the marking scheme, and how edges are mapped during the merge of ape graphs at call sites, it is expected that the actual number of processed edges will be considerably less. For each ape graph edge processed, the algorithm examines each store annotation on the edge exactly once and each load annotation once for each store annotation on the edge.

6. RELATED WORK

This work is related to testing and analysis of object-oriented codes, pointer analysis, and escape analysis.

6.1 Object-Oriented Testing

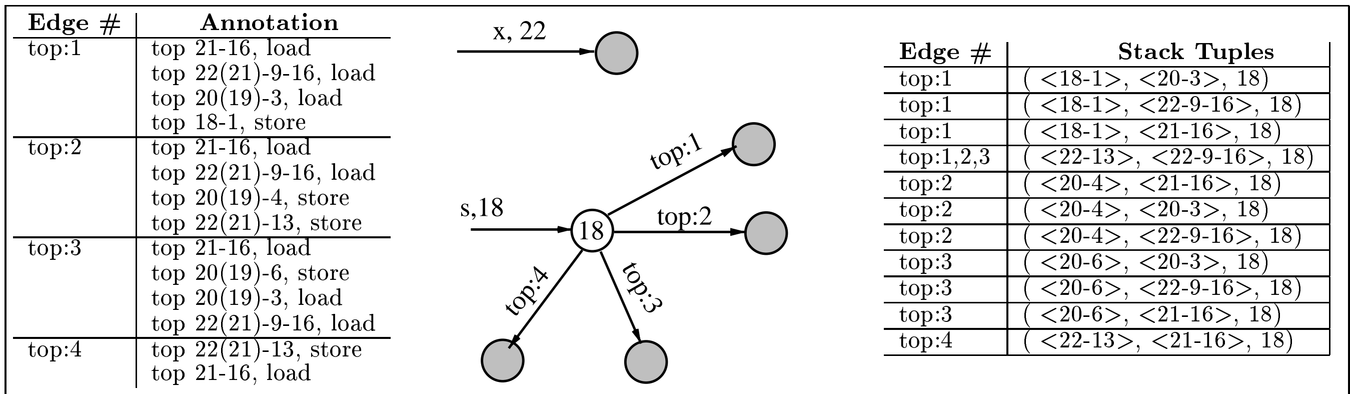
Previous work on structural testing of object-oriented software has concentrated on data flow analysis for computing def-use associations for classes[14], testing of libraries in the presence of unknown alias relationships between parameters and unknown concrete types of parameters, dynamic dispatches, and exceptions[6], and developing a set of criteria for testing Java exception handling constructs[23]. Commercially available tools include a Java testing tool set by SUN which is comprised of JavaSpec, JavaStar, JavaScope, and JavaLoad [26].

Previous research on data flow analysis of classes has focused on intra-method, inter-method, and intra-class def-use pairs of primitive types [14]. Relationships between these def-use pairs and program representations to generate the pairs are defined. The program representations include the class call graph, a frame around the class call graph, and the class control flow graph. The frame represents the unknown sequence of calls from clients by enabling a random calling sequence between methods in a class. The frame also allows techniques for interprocedural def-use analysis to be applied to detect def-use pairs of primitive types in the class with different levels of precision due to aliasing effects and the techniques used for dealing with aliases[15, 21].

We consider this to be the closest work to our work. While this technique finds def-use pairs within a single class, it does not find inter-class def-use pairs in a scalable manner. The program representation would not scale well for programs with a large number of interacting classes. Our approach uses a compositional representation, which does not require the entire program representation to be in memory at once. Another drawback of this approach is the lack of association between objects and their fields in the analyses. Our use of object manipulations easily makes this association, which we feel is critical due to the importance that objects play in object-oriented programs.

Larsen and Harrold[18] introduced the concept of a class dependence graph and adapt the system dependence graph for object-oriented software. Their program representation fully represents the features of object-oriented software including inheritance and polymorphism. They use this program representation for slicing object-oriented software[18]. Liang and Harrold extended the program representation once more for use with object slicing[19].

Chatterjee, Ryder, and Landi introduced relevant context inference (RCI) as a modular technique for flow and context-sensitive data flow analysis of statically typed object-oriented programming languages[7]. Modularity makes it possible to avoid having the entire program representation in memory



Edge #	Annotation
top:1	top 21-16, load top 22(21)-9-16, load top 20(19)-3, load top 18-1, store
top:2	top 21-16, load top 22(21)-9-16, load top 20(19)-4, store top 22(21)-13, store
top:3	top 21-16, load top 20(19)-6, store top 20(19)-3, load top 22(21)-9-16, load
top:4	top 22(21)-13, store top 21-16, load

Edge #	Stack Tuples
top:1	(<18-1>, <20-3>, 18)
top:1	(<18-1>, <22-9-16>, 18)
top:1	(<18-1>, <21-16>, 18)
top:1,2,3	(<22-13>, <22-9-16>, 18)
top:2	(<20-4>, <21-16>, 18)
top:2	(<20-4>, <20-3>, 18)
top:2	(<20-4>, <22-9-16>, 18)
top:3	(<20-6>, <20-3>, 18)
top:3	(<20-6>, <22-9-16>, 18)
top:3	(<20-6>, <21-16>, 18)
top:4	(<22-13>, <21-16>, 18)

Figure 6: Ape graph, annotations and tuples for main with unknown Node class

at the same time. Their approach is capable of analyzing complete systems as well as incomplete systems such as libraries. RCI has been used in the application of data flow based testing of object-oriented libraries[6]. Their technique focuses on the difficulties attributed to testing libraries due to unknown alias relationships between parameters, unknown concrete types of parameters, dynamic dispatches, and exceptions. These difficulties arise from the fact that no driver program exists in library testing. They have developed an algorithm for finding def-use associations in object-oriented libraries.

6.2 Pointer Analysis

There has been a large body of work in pointer analysis, resulting in a set of techniques which vary in the precision of the results and the analysis efficiency. *Flow insensitive* techniques do not consider the control flow within procedures, achieving fast analysis time at the cost of precision loss [22, 25, 1, 16]. A single points-to graph that is valid for the entire program is produced. *Flow sensitive* analysis typically involves solving a data flow analysis problem to obtain the intraprocedural results [12, 17, 8, 30]. An interprocedural analysis is *context sensitive* when it takes into account the legal call/return sequence of procedure calls, whereas a *context insensitive* analysis does not go to the expense of avoiding the inclusion of information along invalid call-return paths. Flow sensitive, context sensitive analyses are very precise, but often require a long time as well as a large amount of space. Liang and Harrold developed a flow insensitive, context sensitive algorithm with the goal of bridging the gap between speed and precision [20].

The concept of invisible variables was developed to parameterize the analysis result for pointer and shape analysis algorithms so that the result can be used at different call sites[17, 12, 30]. The points-to escape graph differs from the invisible variable approach in its distinction between inside and outside edges, and how the relationship between outside objects is determined[29].

6.3 Escape Analysis

Escape analysis has historically been developed and used in the context of functional languages [13, 3, 10]. More recently, escape analysis has been used for optimizing object-oriented programs [29, 9, 4, 5]. Escape analysis identifies objects in the program that do not escape a method and therefore can be allocated to the stack. This reduces the amount of space needed for allocating objects on the heap and also reduces the time spent by the garbage collector deallocating heap space. Unnecessary thread synchronization operations can be eliminated when escape analysis identifies objects that are isolated to one thread.

7. CONCLUSIONS AND FUTURE WORK

This paper presents a strategy for structural testing of object-oriented software systems with possibly unknown clients and unknown information about invoked methods. By exploiting the combined points-to and escape analysis developed for compiler optimization, the *OMEN* testing paradigm does not require a whole program representation to be in memory simultaneously for testing analysis. Potential effects from outside the component under test are easily identified and reported to the tester. The ape graph is easily extended and the computation of additional test tuples can be performed in a demand-driven way as clients are added, without requiring an expensive, predictive computation of test tuples due to potential clients.

We are currently extending the approach to include more information on copies and other manipulations to object references to increase the precision of the analysis. In addition, we are designing a testing tool based on the *OMEN* approach. Lastly, we plan to perform empirical studies to evaluate the *OMEN* approach on large Java software systems focusing on experiments with different scenarios of unknown information about clients and invoked methods.

8. REFERENCES

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

[2] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA*, 1996.

[3] B. Blanchet. Escape Analysis: Correctness proof, implementation and experimental results. In *POPL*, 1998.

[4] B. Blanchet. Escape analysis for object-oriented languages, application to Java. In *OOPSLA*, 1999.

[5] J. Bodga and U. Hoelzle. Removing unnecessary synchronization in Java. In *OOPSLA*, 1999.

[6] R. Chatterjee and B. G. Ryder. Data-flow-based Testing of Object-Oriented Libraries. Technical Report 382, Rutgers U., 1999.

[7] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant Context Inference. In *POPL*, 1999.

[8] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, 1993.

[9] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *OOPSLA*, 1999.

[10] A. Deutsch. On the Complexity of Escape Analysis. In *POPL*, 1997.

[11] A. Diwan, J. E. B. Moss, and K. McKinley. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. In *OOPSLA*, 1996.

[12] M. Emami, R. Ghiya, and L. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *PLDI*, 1994.

[13] B. Goldberg and Y. Park. Escape Analysis on Lists. In *PLDI*, 1992.

[14] M. J. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. In *FSE*, 1994.

[15] M.J. Harrold and M.L. Soffa. Interprocedural Data Flow Testing. In *TAV*, 1989.

[16] M. Hind, M. Burke, P. Carini, and J.D. Choi. Interprocedural pointer alias analysis. *ACM Trans. on Prog. Lang. and Sys.*, 1999.

[17] W. Landi and B. G. Ryder. A Safe Approximation Algorithm for Interprocedural Pointer Aliasing. In *PLDI*, 1992.

[18] L. Larsen and M. J. Harrold. Slicing Object-Oriented Software. In *ICSE*, 1996.

[19] D. Liang and M. J. Harrold. Slicing Objects Using System Dependence Graphs. In *Inter. Conf. on Soft. Maint.*, 1998.

[20] D. Liang and M. J. Harrold. Efficient Points-to Analysis for Whole-Program Analysis. In *FSE*, 1999.

[21] H. Pande, W. Landi, and B.G. Ryder. Interprocedural Def-Use Associations in C Programs. *IEEE Trans. on Soft. Eng.*, 1994.

[22] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *POPL*, 1997.

[23] S. Sinha and M. J. Harrold. Criteria for Testing Exception-Handling Constructs for Java Programs. In *Inter. Conf. on Soft. Maint.*, 1999.

[24] A. L. Souter and L. L. Pollock. Inter-class Def-Use Analysis with Partial Class Representations. In *PASTE*, 1999.

[25] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[26] Sun Microsystems. Java Testing Tool. www.sun.com/workshop/testingtools/index.html, 1998.

[27] C. Szyperski. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[28] A. V.Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.

[29] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *OOPSLA*, 1999.

[30] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *PLDI*, 1995.

[31] S. H. Yong, S. Horwitz, and T. Reps. Pointer Analysis for Programs with Structures and Casting. In *PLDI*, 1999.