

The Construction of Contextual Def-Use Associations for Object-oriented Systems

Amie L. Souter, Lori L. Pollock

Abstract

This paper describes a program representation and algorithms for realizing a novel structural testing methodology that not only focuses on addressing the complex features of object-oriented languages, but also incorporates the structure of object-oriented software into the approach. The testing methodology is based on the construction of **contextual def-use associations**, which provide context to each definition and use of an object. Testing based on contextual def-use associations can provide increased test coverage by identifying multiple unique contextual def-use associations for the same context-free association. Such a testing methodology promotes more thorough and focused testing of the manipulation of objects in object-oriented programs. This paper presents a technique for the construction of contextual def-use associations, as well as detailed examples illustrating their construction, an analysis of the cost of constructing contextual def-use associations with this approach, and a description of a prototype testing tool that shows how the theoretical contributions of this work can be useful for structural test coverage.

Keywords

D.3.2.p Object-oriented languages, D.2.5.m Testing coverage of code, object flow analysis.

I. INTRODUCTION

Application of object-oriented design principles results in programs with a structure that differs significantly from that of the imperative programs originally targeted by control flow and data flow-based testing methods. The novel characteristics of object-oriented software are primarily due to classes, inheritance, polymorphism and dynamic binding. In an object-oriented domain, a class becomes a natural unit of programming, compilation, debugging,

and unit testing. However, 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.

This paper describes a program representation and algorithms for realizing a novel structural testing methodology that focuses on addressing not only the complex features of object-oriented languages, but also the structure of object-oriented software. The testing methodology is based on the construction of *contextual def-use associations*, which provide context for each definition and use of an object. This novel formulation of individual object definitions and uses seeks to capture object aggregation in addition to addressing inheritance and polymorphism. An early formulation of this approach was described in [15].

Programmers create object aggregation relations when they design a class that includes one or more objects of other classes. The resulting object def-use associations are *contextual* in that they provide context through a sequence of method calls, with the computed def-use associations in an object-oriented program. This is in contrast to traditional def-use associations in which a variable v is an ordered tuple (v, d, u) where d is a statement defining v and u is a statement that is reachable by at least one definition-clear path from d , and u uses v or a memory location bound to v . Such a def-use association is *context-free* because it is reported free of any context.

One useful application for contextual def-use associations is structural testing coverage that captures the object aggregation relations. Testing based on contextual def-use associations can provide increased test coverage by identifying multiple unique contextual def-use associations for the same context-free association. In addition, through contextual def-use associations based on object aggregation relations, selected combinations of method invocations can be targeted for testing. The impact is more thorough and focused testing to be

performed for the manipulation of objects in object-oriented programs.

Previous work enables data flow testing of primitive type variables and has started to address the issues with aggregate relationships [11], [1], [5], but no existing structural testing technique automatically generates calling sequences associated with aggregate object relationships. This paper demonstrates that the construction of these method call sequences is feasible and practical, through the use of a program representation that models the relations between objects. This methodology has been incorporated into a **T**esting and **A**nalysis **T**ool for **O**bject-**O**riented programs, TATOO, which establishes an environment for systematically testing object-oriented programs.

Relevant background pertaining to structural testing of object-oriented programs is described in Section II. In Section III, the key object manipulations on which this testing methodology is based are described. Section IV describes the program representation and the extensions for constructing contextual def-use associations (cdus). Section V presents the algorithm that performs the actual construction of cdus with an example presented in Section VI. A description of how to construct different levels of cdus is provided in Section VII. More details on different levels of cdus and their properties are given in [16], [14]. Section VIII presents an analysis of space and time costs for constructing cdus, and empirical results that illustrate the cost of this technique. The TATOO testing environment is presented in Section IX. Finally, a summary and future directions are presented in Section X.

II. STRUCTURAL TESTING OF OBJECT-ORIENTED SOFTWARE

Structural testing research for object-oriented software has investigated data flow testing of classes, automated class testing, integration testing, and library testing in the presence

of unknown alias relationships [7], [17], [1], [5], [4], [3]. Harrold and Rothermel [7] defined the relationship between intra-method, inter-method, and intra-class def-use pairs of fields and local variables of primitive types, and provided program representations (i.e., the class call graph, class control flow graph (CCFG), and framed CCFG), to enable identification of these def-use pairs in a context-free manner.

Buy et al. [3] developed a technique to automate class testing, based on producing sequences of method invocations that can be used to exercise the class under test. This technique uses data flow analysis on the CCFG to calculate context-free def-use pairs for each instance variable of a class. Kung et al [10] also developed a class testing technique, which generates a sequence of method invocations, but relies on calculating a state-based model, which represents the possible states of each primitive type instance variable through the use of finite state machines.

Integration testing techniques for object-oriented software have been studied from both the black-box [9], [12], [10] and white-box testing perspective [11], [1], [5]. Orso [11] 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-use to take into account polymorphism, but are context-free. Alexander and Offutt also developed an integration testing technique for object-oriented programs by extending their method coupling technique [8] to handle inheritance, aggregation, and polymorphism [1]. Individual definitions and uses are context-free.

Chen and Kao developed an object flow-based testing strategy to help guide test case generation [5]. The all-binding criterion is used to ensure coverage based on exercising all possible run time types of an object that occur due to inheritance and polymorphism. The

all-du-pair criterion is used to monitor the behavior of objects by keeping track of where an object is defined and used. These techniques do not address aggregation.

An extension to Harrold and Rothermel’s data flow testing of classes was investigated by the authors in [17]. The focus was on how to compute *inter-class def-use pairs* in the presence of interacting classes with many methods and complex interactions. This work led to the concept of contextual def-use pairs.

In summary, existing techniques address the novel characteristics of object-oriented software primarily due to classes, aggregation, inheritance, polymorphism, and dynamic binding. However, programming style and units of analysis for object-oriented programs also differ from imperative programs. In addition, the context in which def-use pairs are typically reported provides no context in terms of how an object is currently used. For example, def-use pairs of instance variables are not typically reported associated with an encapsulating object, nor do they provide context in terms of the sequence of calls that an invoking object executed in order to reach the def and use of instance variables of the object.

Previous approaches also fall short in providing the tester feedback regarding the testing of incomplete programs. Buy et al. [3] provide feedback in terms of def-use pairs, when symbolic execution fails to complete, but this feedback does not indicate how objects interact with other objects. Similarly, there are no approaches that inform the tester of the fact that an object interacts with an unanalyzed region of a program and could potentially be modified, indicating that an external source could be affecting the testing result.

In this paper, a structural testing methodology based on the notion of contextual def-use associations is presented; context is provided to each definition and use of an object. Context is captured through a sequence of method calls from the invoking object to the definition and

to the use. By extending the points-to escape graph representation developed by Whaley and Rinard [19], we can identify object creation sites and potential writes and reads to the same object for testing, similar to data flow testing of primitive type variables, in a straightforward way.

III. OBJECT MANIPULATIONS

It is natural for an object-oriented testing technique to focus on object manipulations in order to expose the possible behaviors of object-oriented software. In this paper, we define the concept of contextual def-use associations (cdus) which provide the basis for a testing methodology based on object manipulations. Contextual def-use associations are defined as a tuple (o, def, use) for an object o in which the def and use are each defined to be a call sequence $(CS_{om}-CS_1-\dots-CS_m-L)$ where CS_{om} is the call site of the first method call leading to a modification (reference) of the state of object o . L is the location of the actual store (load) setting the modified (referenced) state for object o . Each CS_i in the internal sequence is a call site in a call sequence leading from the original call site, CS_{om} , to the store (load) L .

The most elemental object manipulation is identified as either a read or write operation. The actions that a particular statement or method perform on an object can be decomposed into a sequence of these elemental operations. In particular, eight elemental read and write operations have been identified [19] as follows, read or write the value in a reference to an object, read or write the value in a field of an object, create or delete a new object, pass a reference to an object as a parameter, or return a reference to an object from a method.

These 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, a *set* of objects potentially may be referenced by each reference. For these descriptions, singular form is used; however, the points-to escape analysis addresses the potential for a set of objects to be referenced. Note that any time a reference is changed, the reference to the previous values are killed.

- *Copy*, $r_1 = r_2$, sets reference r_1 to point to the object referenced by r_2 ; r_1 no longer points to the object it previously referenced (unless it was also referenced by r_2).
- *Load* statement, $r_1 = r_2.f$, sets reference r_1 to point to the object referenced by field f of the object referenced by r_2 .
- *Store* statement, $r_1.f = r_2$, sets field f of the object referenced by r_1 to point to the object referenced by r_2 .
- *Global load*, $r = cl.f$, sets r to point to the object referenced by the variable f of class cl .
- *Global store*, $cl.f = r$, sets variable f of class cl to point to the object referenced by r .
- *Return* statement, *return* r , indicates that the method in which the return statement is located returns the object referenced by r .
- *Object creation*, $r = new\ Object(...)$, an object of class *Object* is created, and r is set to point to the new object. Parameters are handled analogous to other method invocations.
- *Method invocation*, $r = r_0.methodname(r_1, r_2, \dots, r_n)$ results in invoking the method called *methodname* of the object referenced by r_0 , passing object references r_1, r_2, \dots, 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.

Transforming source code into a form that represents object manipulations is straightforward. For example, the FLEX compiler framework [13], which is the infrastructure used in

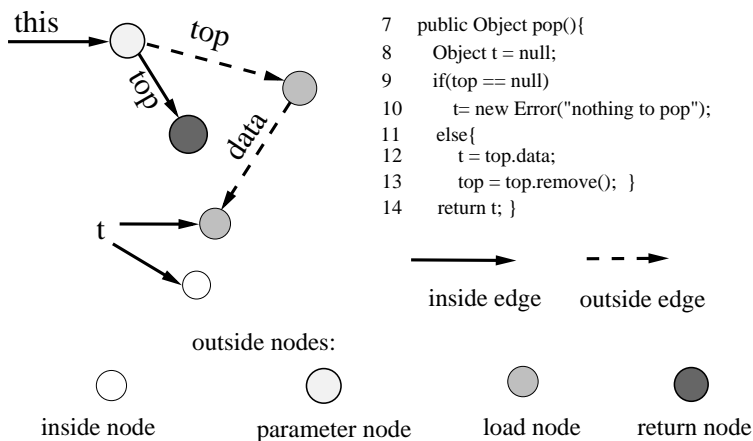


Fig. 1. Example of complete points-to escape graph for a single method.

this work, transforms each Java bytecode instruction into an intermediate code statement that naturally expresses one of the above basic statements.

IV. PROGRAM REPRESENTATION

In this work, the relationships between objects and their fields are represented by an *APE* graph, which is an extension to the points-to escape graph representation developed by Whaley and Rinard [19].

A. The Points-to Escape Graph Representation

Definition. 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 [19]. 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 that are either callers or callees of the current analysis region. 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.

```

class Stack{
    Node top;
1 public Stack() { top = null; }
2 public void push(Object e) {
3     if (top == null)
4         top = new Node(e, null);
5     else
6         top = top.insert(e); }

17 public static void main(string args[]){
18     Stack s = new Stack();
19     for(int i=0; i < 10; i++)
20         s.push(new Integer(i*2));
21     while(!(s.isempty())){
22         Object x = s.pop();
23         System.out.println(x); } }

class Node{
    Object data;
    Node next;
24 Node(Object e, Node n) {
25     data = e;
26     next = n; }
27 Object get() { return data; }

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

28 Node insert(Object e){
29     Node temp = new Node(e, this);
30     return temp; }

31 Node remove(){
31     Node n = this;
32     n = n.next;
33     return n; }

```

Fig. 2. Example: Stack and Node classes.

Whaley and Rinard [19] developed this representation to enable heap and synchronization optimization of Java programs. 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 is flow sensitive and analyzes each method once to produce a single parameterized analysis result that allows for analyzing a method independent of its callers and unanalyzed callees, and then combines analysis results from multiple methods through interprocedural analysis. The precision of the results is increased as invoked methods are analyzed.

In a points-to escape graph, nodes represent objects that the program manipulates and edges represent references between objects. Figure 1 illustrates a complete points-to escape

graph for a single method as the current analysis region. A node is either an inside or outside node. Each *inside node* represents an object creation site for objects created and reached by references created inside the current analysis region. A single inside node representing an object creation site represents all objects created at that site. In Figure 1, an inside node is shown that represents an instance of an `Error` object pointed to by `t`. A *class node* represents a single class. There is one class node for each class in the program. The fields in this node represent the static class variables corresponding to the node's class.

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

Each load statement (e.g., access as `v.f` for field `f` of object `v`) 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 statement `top == null` generates a load node pointed to by `top`. Finally, there is one *return node* for each method invocation site in the method to represent the return values of unanalyzed methods. In Figure 1, method `remove` is assumed to be unanalyzable; the reference `top` points to a return node. In contrast to a return node, `t` points to the possible return values of method `pop`, which represents the return statement of `pop`. The reference representing the return value points to all nodes that could possibly be returned by the method.

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 (i.e., the current analysis region *reads* a reference created in an unanalyzed region). Thus, each outside edge points to a load node. This is illustrated in Figure 1 with the references **top** and **data** pointing to load nodes.

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. An *escaped* node represents the fact that the object escapes to unanalyzed portions of the program. An object can directly 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. In Figure 1, the return values pointed to by **t** escape the current analysis region, which is the method **pop**. Though the object escapes this method, it may be recaptured by a caller of the method **pop**.

Construction. The detailed algorithm for points-to escape graph construction is given in Whaley and Rinard’s paper [19]. This section presents an example of using the construction method in order to aid in understanding the graph’s meaning, the handling of interprocedural analysis, and extensions for the *APE* graph.

We first build a conservative call graph using the RTA construction algorithm [2]. Because

points-to graphs for callees are merged into their callers' graph, non-recursive programs are processed in a reverse topological sort over the call graph. Recursive programs involve fixed-point iterative analysis within each strongly connected component of the conservative call graph.

For a given method, an initial points-to escape graph is first constructed to represent only the parameters and class objects on entry to the method. This points-to escape graph is refined by repeatedly processing only the object-related statements in the method's control-flow graph (CFG) until a fixed point is reached.

At each of the object-related statements s , the points-to escape graphs representing each of the predecessor statements of s in the CFG are first merged into a single graph. The new points-to escape graph in effect at the program point immediately after s is constructed by applying s 's transfer function to the merged graph that was in effect immediately before s .

When a method call site is encountered during the construction of a points-to escape graph, interprocedural analysis is achieved through merging the parameterized points-to escape graphs of all the potentially invoked methods 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 example, if a call of the form `x.insert(...)` is encountered at statement w in method y when the graph for y is being built, the graph just prior to w in y , and the graph for all potentially invoked `insert` methods (due to polymorphism) are merged to form the graph after statement w . For call sites to unanalyzed methods, the parameters and return value are marked as escaped within the caller's graph.

Figures 3 and 4 show the steps of the construction of the points-to escape graph for method `push` of the `Stack` class shown in Figure 2. In this case, the call graph stemming

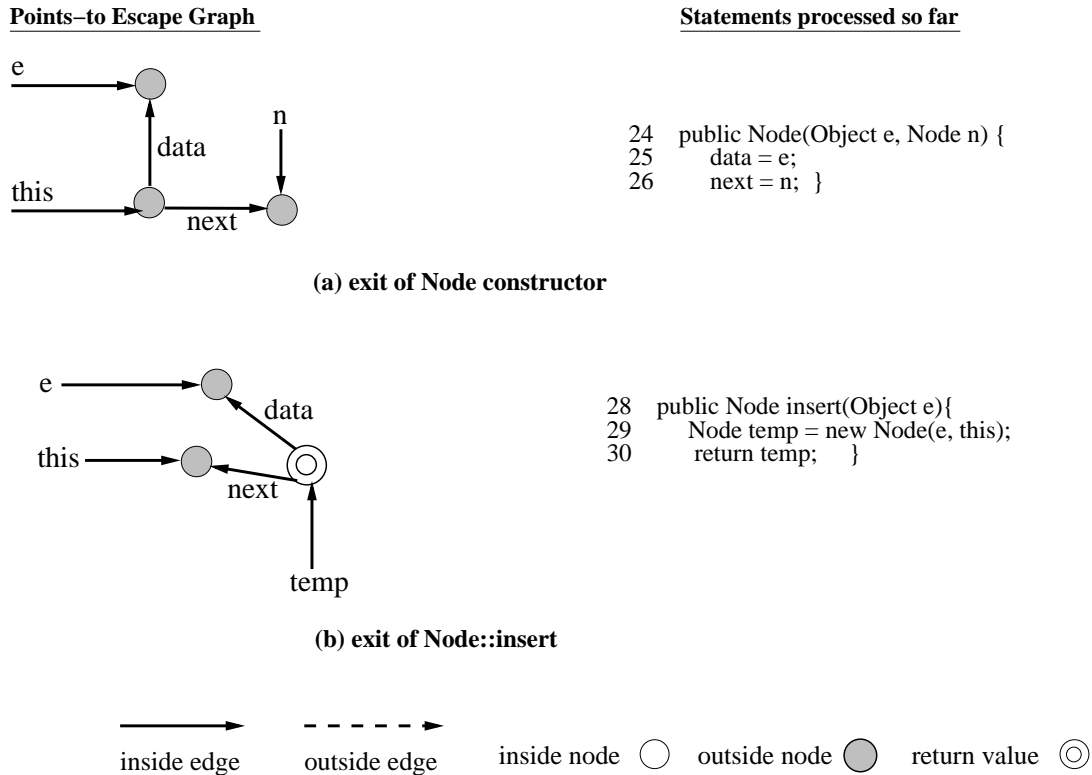


Fig. 3. Points-to escape graph example

from `push` consists of edges to nodes for `Node`'s constructor and `Node`'s `insert` method. Following the reverse topological order of the call graph, the points-to escape graphs for `Node`'s constructors and the `insert` method are constructed first.

The points-to graph for `Node`'s constructor is created first. Nodes representing each parameter in the method including the receiver object `this` are constructed. All of the nodes are outside nodes due to the fact that the objects they represent were created outside the current analysis region, the `Node` constructor. Each statement in the method is processed and edges relating to each object manipulation are added to the graph. Figure 3(a) illustrates the points-to escape graph for the exit of the `Node` constructor.

Next, `insert`'s graph is constructed. The process begins by creating nodes for each parameter, namely `e` and `this`. Next, each statement is processed, resulting in the reference `temp`

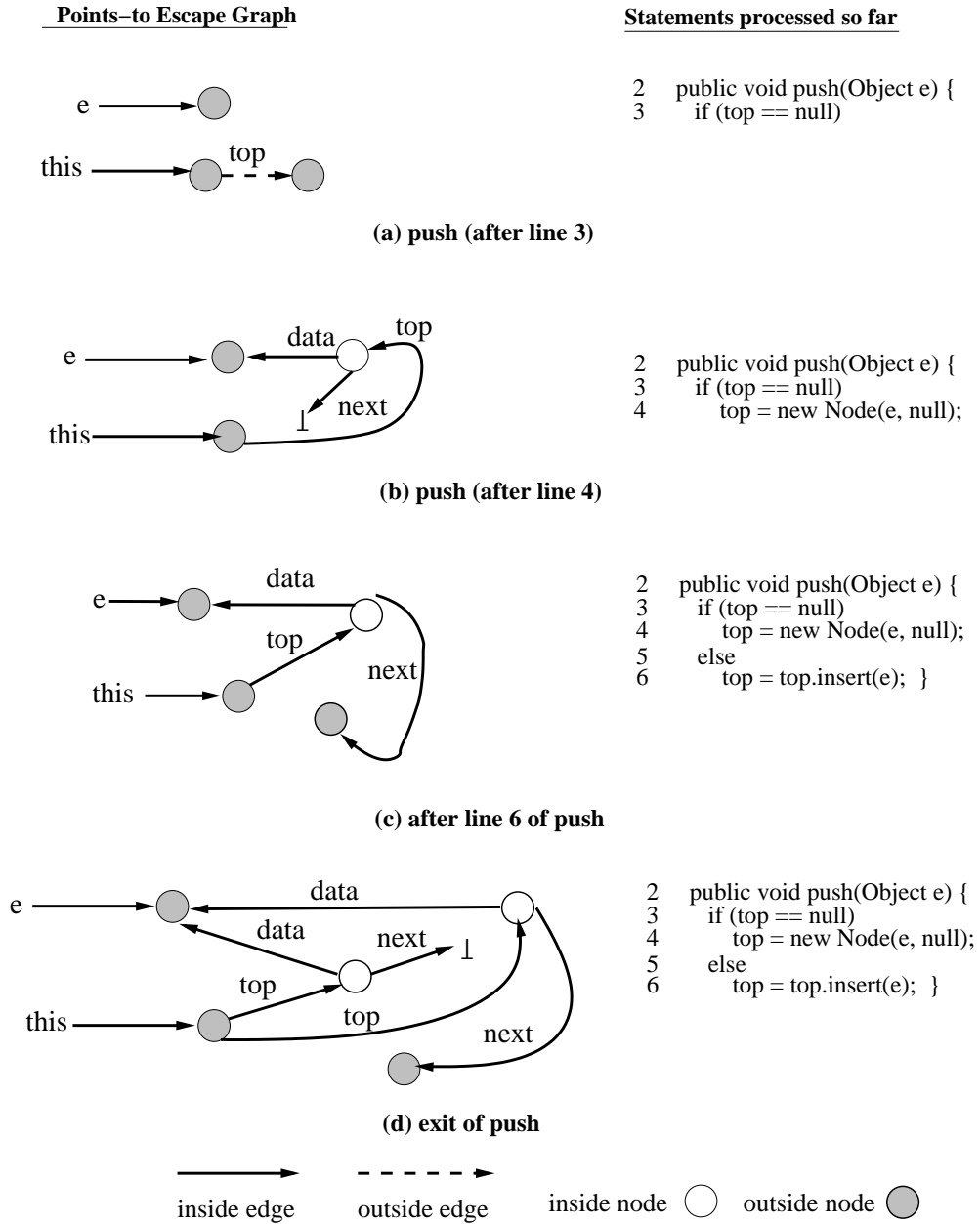


Fig. 4. Points-to escape graph example (continued).

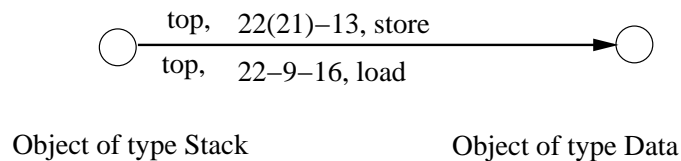


Fig. 5. Illustration of *APE* graph annotations.

pointing to an inside node (i.e., an object created inside the current analysis region, `insert`), with an edge labeled by the `data` field pointing to `e` and an edge labeled by the `next` field which points to the node representing the object referenced by `this`. The nodes of the `Node` constructor's graph are mapped to nodes in `insert`'s graph and then edges are matched. The inside node referenced by `temp` is also the return value of the `insert` method, as indicated by the double circle. Figure 3(b) illustrates the graph for the exit of `insert`.

After processing the callees of `push`, `push`'s graph is constructed. 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. Figure 4(a) illustrates the graph at line 3 of `push`, where the object manipulation associated with the load of `top` is represented in the graph by adding an edge from the node representing `this` to a newly constructed node representing the object associated with the `top` field. Since the object that `top` references was created outside the scope of this method, the new node is an outside (load) node, and an outside edge for `top` points to the new load node.

Line 4 in method `push` corresponds to an object creation site, which is similar to a method invocation in that we must first retrieve the graph associated with the object creation site, which in this case is the graph associated with the `Node` constructor. To perform the interprocedural merge of the `Node` constructor's graph into `push`'s graph at line 4, the nodes representing the formal parameters in the `Node`'s constructor graph are mapped to the nodes corresponding to the actuals in `push`, and then outside edges of the callee are mapped to inside edges of the caller. Figure 4(b) illustrates the graph of `push` after merging in the graph for `Node`'s constructor. In this example, the nodes for the formal parameters of `Node` are `this`, `e`, and `n`. These nodes are mapped to the actual parameters in the call to `Node`

in `push`, which are the nodes pointed to by `top`, `e`, and `null`, respectively. Then, the inside edges of `Node` associated with the nodes that map to `push` are mapped to the corresponding edges in `push`. This results in a node for a new object with an edge labeled `data` pointing to the node pointed to by parameter `e`, an edge labeled `next` pointing to `null`, and an edge labeled `top` pointing to the new node.

The next statement in method `push` is the `else` branch, which is another method invocation. The points-to escape graph associated with the callee `insert` is retrieved. The process of mapping the nodes representing the formals of the callee to the actuals of the caller and inside edges of the callee to become newly constructed outside edges of the caller is performed to merge `insert`'s graph into `push`'s graph at line 6. When the graph in Figure 3(b) is merged into the graph in Figure 4(a), a graph for `push` after line 6 is obtained, illustrated in Figure 4(c). The reference `top` points to an inside node that corresponds to the return value of the method `insert`. The mapping of nodes from `insert` to `push` results in the edge labeled by the `data` field of `top` pointing to the node representing the object referenced by parameter `e`, and the edge labeled by the `next` field pointing to the object referenced previously by `top`, which is represented by the new outside node that no other nodes are referencing.

Finally, the points-to escape graphs of Figure 4(b) and 4(c) 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`, shown in Figure 4(d).

B. The APE Graph

Motivation and Definition. The Annotated Points-to Escape (APE) graph¹ is an extension of the points-to escape graph to enable the formulation of `cdus`. The set of points-to escape

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

graphs for a program provides an object-based program representation that models object manipulations and aggregate relationships occurring throughout the program. In order to construct cdus, additional information about the location and kind of object manipulations needs to be maintained.

The 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 writes to this object prior to this program point?* While Whaley and Rinard [19] compute the points-to escape graph at each program point during their analysis for compiler optimization, only the points-to escape graph for the exit of each method is maintained for the construction of cdus for software testing. It is desirable to avoid unnecessary storage requirements for a points-to graph representation at every program point during testing analysis. The following modifications to the points-to escape graph designed by Whaley and Rinard [19] enable point-wise information to be retained in a single representation for each method.

First, edges are not deleted when statements that alter the references are processed. Second, the labels on edges are replaced by more comprehensive information about the loads and stores associated with the edge's reference. A flow insensitive analysis is performed. By retaining edges and augmenting the annotations, it is possible to compute cdus with only a single points-to escape graph per method. Recall that outside edges labeled by fields are only created by processing load statements, and inside edges labeled with a field are only created by store statements. However, an inside edge of an *APE* graph can 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 . Multiple annotations can be added to edges in two different places in the construction of an *APE* graph, namely control flow join points and the interprocedural merge of *APE* graphs.

In order to report the location of a def of an object or a use of an object, for a def-use association, the location of the store (load) of the reference represented by a particular edge must be maintained. In addition, to report the call sequence for the context of a def or use in a *cdu*, the call sequence leading to the store (load) must be maintained as part of the edge. Thus, each annotation associated with an edge in the *APE* graph contains a sequence of statement numbers, $(s_1 - s_2 - \dots - s_n)$, where s_n is the unique statement number of the load (store) statement; $s_1 - s_2 - \dots - s_{n-1}$ consists of the statement numbers of call sites where this edge was merged into a caller's *APE* graph during interprocedural merge. Statement s_1 is the statement number of the call site within the current analysis method which eventually leads to the load (store) statement. The full statement sequence ending with the location of the store or load is called the call sequence that provides the context for the store (load). In this paper, statement numbers refer to source lines for readability, but actually they are implemented by a mapping between bytecode and source code.

In addition, each store annotation on an edge includes a corresponding sequence of statement numbers, $(evs_1 - evs_2 - \dots - evs_n)$, where each evs_i is the unique number of the earliest statement in the method at which the store at statement s_i could be referenced by other statements. We call this the earliest visible statement, evs_i for s_i . Let G be the control flow graph (CFG) containing s_i , then the earliest visible statement evs_i is the header of the maximal strongly connected component (SCC) in G containing s_i , where the header node

dominates all other nodes in the SCC ². The `main` method in Figure 2 is used to illustrate the need for an *evs* for a given statement. Statement 20 has an *evs* of 19 due to statement 19 being the loop header dominating statement 20. Intuitively, a store in a loop could affect statements lexically before the store, but not lexically before the header of the loop. The call sequences provide the context of loads and stores, while the *evs* for stores enable correct determination of the order of manipulations to objects.

Statement number sequences also label nodes to indicate the statement that created the node and the calls leading to that statement from the current analysis method. No additional annotations are needed to handle recursion; the annotations added during the interprocedural merge operation will suffice to identify *cdu* pairs involved in recursion.

Figure 5 shows an example set of annotations on a single edge of an *APE* graph. The nodes in this graph represent objects created within the current analysis region; therefore, they are both inside nodes. The edge labeled `top` represents a reference from a field named `top` of the object of type `Stack`. The annotations indicate that there exist both a load at statement 16 and a store at statement 13 of the field `top`. The annotation `(22(21)-13 store)` represents one call site on line 22, which leads to a store of `top` at line 13. The number in parentheses represents the *evs* associated with statement 22, indicating that statement 22 occurs inside a loop with the loop header at statement 21. The statement numbers that do not have corresponding *evs* numbers indicate that these statements do not occur within a loop and the *evs* number equals the statement number. Similarly, the load of the field `top` occurs at line 16, following a chain of calls from lines 22 to 9.

Because the *APE* graph construction does not delete edges caused by reassignments to

²If s_i is not in an SCC, no evs_i is shown in the figures

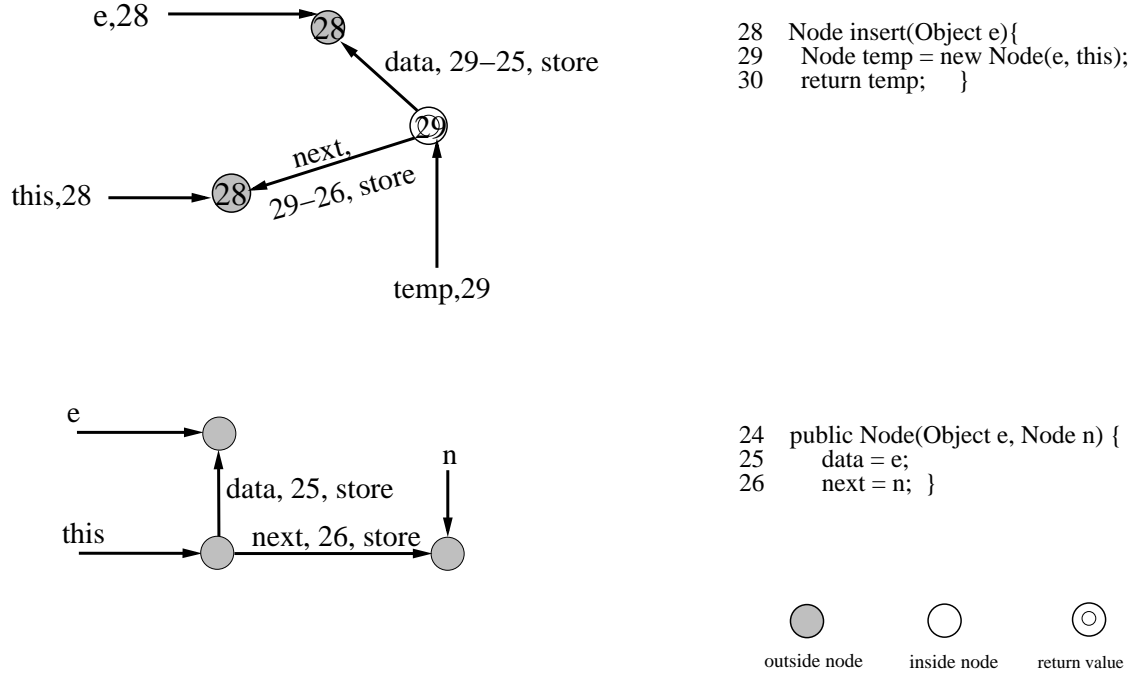


Fig. 6. Illustration of annotation construction.

an object, a loss in precision about points-to relations holding at given points can occur. A solution to this problem would be to maintain kill information on edges to reflect statements where a reference would be killed and removed by the points-to escape graph. This information would require more space, but would provide more precise results. The imprecision of points-to escape graphs and therefore the *APE* graphs will impact the set of generated cdus in particular, causing some infeasible cdus to be constructed.

Construction. The annotation for a given load (store) is incrementally constructed as the *APE* graph is formed for a given method by modifying Whaley and Rinard’s points-to escape graph merge algorithm [19] performed at a call site. 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 evs for the call site are concatenated onto the front of the annotations labeling the corresponding edges from the callee’s *APE* graph. Algorithm 1 presents the interprocedural merge algorithm extended with lines 10-12 to incrementally construct annotations. Figure 6

illustrates the formation of annotations. The `data` and `next` edges of the `Node` constructor’s graph are labeled to indicate that stores have occurred on lines 25 and 26, respectively. During the merge of the `Node` constructor’s graph with the `insert` graph at line 29, the nodes and edges of the `Node` constructor’s graph are mapped to nodes and edges in the `insert` graph. In addition to the edge and node mapping, the annotations of the callee graph are augmented to form the annotations for the caller’s graph. As the `insert` graph annotations show, the call site at line 29 is concatenated to the callee’s annotations to form the caller’s annotations.

Algorithm 1 Extended Interprocedural Merge Operation.

Input: *APE* graph A_{before} before callsite at statement number s in caller;

APE graph A_{callee} for exit of callee;

Output: *APE* graph A_{after} representing A_{before} merged with A_{callee} ;

```

1: let the nodes of  $A_{after}$  = nodes of  $A_{before}$ ;
   /* map callee's nodes into  $A_{after}$  */
2: map each formal parameter node of  $A_{callee}$  to corresponding actual parameter node of  $A_{before}$ ;
3: foreach load node of  $A_{callee}$  that is escaped in caller do
4:   let there be a load node in  $A_{after}$ ;
5: foreach inside node of  $A_{callee}$  reachable in the caller do
6:   let there be an inside node in  $A_{after}$ ;
   /* map callee's edges with caller's edges, building annotations*/
7: foreach edge  $e$  connecting pairs of nodes present in both  $A_{after}$  and  $A_{callee}$  do
8:   if  $e$  does not already exist in  $A_{before}$  then
9:     add  $e$  to  $A_{after}$ ;
   /*extension to build annotations*/
10: if cdu level specifies annotation at this callsite then
11:   concatenate  $s$  to each load annotation on  $e$ ;
12:   concatenate  $s$  and  $evs_s$  to each store annotation on  $e$ ;
13: mark edge  $e$  in  $A_{callee}$  to indicate mapped to caller;
```

The points-to escape graph merge is also extended to mark each callee’s *APE* graph edge that is mapped into a caller’s *APE* graph. After an *APE* graph of a method has been mapped to possibly several caller’s *APE* graphs, the *APE* graph for the given method will have all edges marked which have been mapped to any caller’s graph. All unmarked edges in a method’s *APE* graph are not reachable by callers. These marks are used in the test tuple construction to avoid creating duplicate test tuples for callers and callees and to identify

loads that may have reaching stores outside the component under test.

V. TEST TUPLE CONSTRUCTION

The algorithm for calculating contextual def-use associations for the component under test (CUT) is shown in Algorithm 2. Given the call graph for the CUT and associated *APE* graphs as input, this algorithm computes a set of cdus for the CUT, where a component is any set of methods to be tested. For testing, the computed cdus are augmented with an additional field, namely the object creation site of the associated object. Therefore, the algorithm constructs test tuples of the form (object-name, object creation site, def, use), where object-name is the name of the object created at the creation site.

The general algorithm starts at the root of the call graph for the CUT and proceeds to process each method in a topological order over the call graph nodes³. Each method’s *APE* graph is analyzed once during the traversal over the call graph. By extending the merge operation of the *APE* graph construction algorithm to mark all edges in a callee’s graph that are merged into its caller’s *APE* graph, and processing the call graph in topological order during the generation of cdus, the calculation of cdus from merged subgraphs of the callee will not be performed again. As a particular *APE* graph is analyzed, only unmarked edges (those not already processed in a caller’s graph) are processed in the callee’s graph.

The algorithm examines 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 cdus are created. The annotations reflect the results of the flow insensitive analysis used to build the *APE* graph. Thus, the *evs* and *cs* statement numbers of these annotations are used to identify the reachable loads from a store. In particular, if

³Note that a CUT could have more than one root, for example, with a reusable component library; in such a case, the techniques would work similarly for each rooted call graph.

Algorithm 2 Compute test tuples for an object-oriented component.

Input: a call graph 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;

```

1: Let T be a topological ordering of the nodes in the call graph for CUT, starting at the root of the call graph;
2: foreach node n in T do
3:   Let  $AG_m = APE$  graph for method m represented by node n;
   /* create tuples from stores in m or stores reachable from m */
4:   foreach unmarked edge e in  $AG_m$  labeled by a STORE do
5:     foreach STORE s labeling e do
6:       Let  $cs_s =$  unique statement number for s in  $AG_m$ ;
7:       Let  $evs_s =$  earliest visible statement for s in  $AG_m$ ;
8:       Let  $f_e =$  field labeling e;
9:       foreach LOAD l annotation on e do
10:        Let  $cs_l =$  unique statement number for l in  $AG_m$ ;
11:        if  $evs_s < cs_l$  then
12:          Create a tuple of the form (store,load)
          where store = ( $cs_s$ -currently forming statement sequence for s),
          and load = ( $cs_l$ -currently forming statement sequence for l);
          /*find object creation site for this store-load pair*/
13:          Let  $sn =$  source node of e;
14:          Let  $cs_{sn} =$  unique statement number for  $sn$  in  $AG_m$ ;
15:          if  $sn$  is an inside node then
16:            Replace tuple (store,load) by ( $f_e, cs_{sn}, store, load$ );
17:          else /* $sn$  is an outside node*/
18:            if n is a root in a call graph for CUT then
19:              LogFeedback(object for (store,load) is potentially created outside CUT);
20:            else /*n is an interior node in call graph*/
21:              if  $sn$  is not a parameter node then
22:                LogFeedback(object for (store,load) is potentially created outside CUT);
23:              Let  $tn =$  target node of e;
24:              if  $sn$  not escaped and  $tn$  is escaped then
25:                LogFeedback(value loaded in  $cs_{sn}$ )
                is potentially changed by method outside CUT,
                but l is indeed referencing object created at  $cs_{sn}$ ;
          /*examine loads in m or reachable from m*/
26:          foreach unmarked edge e in  $AG_m$  labeled only by LOAD do
27:            Let  $tn =$  target node of e;
28:            if  $tn$  is a load node in  $AG_m$  then
29:              LogFeedback(object for (store,load) is potentially created outside CUT);
30:              LogFeedback(load at statement  $cs_l$  in method m has potentially reaching references from outside CUT);

```

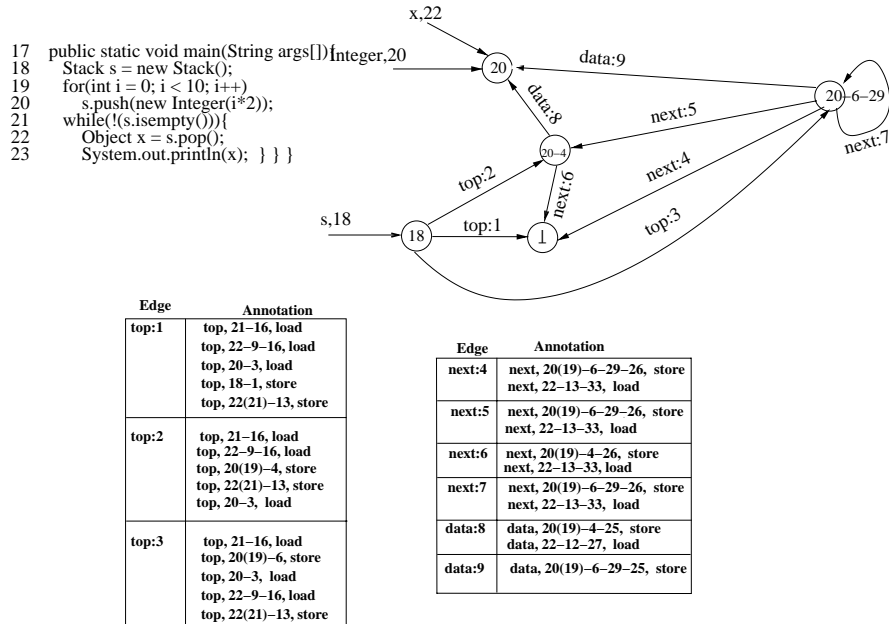
the *evs* of the store is less than the *cs* of the load then the store occurred before the load, and a test tuple for the store and load are created. If the *evs* is greater than the *cs* of the load, then the load does not occur after the store, and a test tuple is not formed. The object creation site 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 represents 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 the CUT, and feedback is logged. This feedback provides information depending on the kind of source node and whether the current method represents an interior or root node of the call graph.

Additionally, feedback is provided when the target node of the *APE* graph edge being analyzed is escaped from the CUT. The algorithm also provides feedback for load nodes when a corresponding store is not present in the CUT. This is indicated 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 the CUT, as well as the possibility that the load in the CUT has potentially reaching references from outside the CUT. One example use of this feedback would be for testing of a user program with a third party component.

VI. EXAMPLE OF CDU CONSTRUCTION

Figure 7(a) shows the source code and complete *APE* graph for the exit of `main`. Edges of the *APE* graph are labeled with a key that indexes into the annotation table below. The root method, `main`, of the call graph is processed first. There are no marked edges in `main`, so all edges are processed. To construct test tuples, reachable loads from each store are determined. The test tuples are formed by finding all loads on the edge that occur after each store. After processing each edge in `main`'s *APE* graph, the next *APE* graph in the topological ordering of the call graph nodes is processed. In this case, `push` is processed, and each *unmarked* *APE* graph edge is analyzed in order to calculate the test tuples. In this case, all edges are marked because all of `push`'s edges have been merged into the caller (`main`'s) graph. Therefore, the test tuples that cover `push` are calculated by processing `main`'s *APE* graph. The `main` program provides the calling context for `push`, as well as the invoking

(a) *APE* graph and edge annotations for exit of main.

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

(b) Computed tuples for main.

Fig. 7. *APE* graph example.

object that establishes context for the instance variable `top` to be manipulated.

To illustrate test tuple construction, the table in Figure 7(b) shows the set of all tuples calculated using Algorithm 2 for the *APE* graph in Figure 7(a). Only the tuples identified when processing `main` are shown. The first column indicates the *APE* graph edge responsible for the testing tuples. The second column shows the tuples in the form (object-name, object

creation site, store, load). For example, the edges `top:1`, `top:2`, and `top:3` create tuples due to the objects created at line 18 in Figure 7(a). The tuple $(\text{top}, 18, \langle 20-4 \rangle, \langle 21-16 \rangle)$ 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 a value loaded at line 16 through the call site at line 21. There is no feedback because the example component is a complete program, and no outside influences manipulate the objects in this program.

VII. ENABLING DIFFERENT LEVELS OF CONTEXT

We developed several strategies for varying the amount of context provided in terms of the call sequences reported from the invoking object to the load or store of the object’s field. The call sequences that provide context for def-use pairs of an object may or may not be a *complete* call sequence which includes every call site along the call chain from the first call to the load (store). Instead, context may be expressed by a *partial* call sequence. A low context level `cdu` includes a partial call sequence with fewer call sites reported along the chain than a higher context level `cdu` which includes a more complete call sequence.

In particular, we defined four levels of context for computing `cdus` based on object aggregation. The naming convention **`cdu-x`** is used to indicate the strategy that computes `cdus` with context level `x`. Higher context levels provide more context with the def and use, but with the tradeoff of increased space (and possibly computational time) and resource requirements for running tests.

Specifically, `cdus` are represented as follows. First, `cdu-0` is a context-free def-use association of the form $(o, \text{def}, \text{use})$. `cdu-1` is a tuple $(o, \text{def}, \text{use})$ for an object `o` in which the `def` and `use` are each defined to be a pair $(\text{CS}_{om}\text{-L})$, where CS_{om} is the call site of the method call leading to a modification (reference) of the state of object `o`, and `L` is the location of the

actual store (load) causing the modified (referenced) state for object o . For $cdu-2$, the pair is replaced by a call sequence of the form $(CS_{om}-(CS_{sccentry}-CS_{sccexit})^*-L)$, where the first and last entries of the sequence are the same as $cdu-1$. The internal sequence is a sequence of pairs of entry and exit nodes of strongly connected components in the call graph. Finally, in $cdu-3$, the call sequences take the form $(CS_{om}-CS_1-...-CS_m-L)$ where the first and last entries of the sequence are the same as $cdu-1$, and each CS_i in the internal sequence is a call site in a call sequence leading from the original call site, CS_{om} , to the store (load) L . In the case of a strongly connected component (SCC) in the call sequence, composed of more than one node, a single sequence of call sites through the SCC is represented.

To enable different context levels for $cdus$, only the incremental construction of annotations during interprocedural merge needs to be changed. In particular, the construction of annotations during the merge operation differs depending on the context level specified. Since the analysis of each callee APE graph occurs before the caller's APE graph, annotations for each callee have been previously formulated when analysis of the caller occurs (i.e., loads and stores have been identified). Therefore, since the merge occurs at every call site, we can modify how call sites are concatenated to the annotations propagated from the callee's graph. For example, for $cdu-1$ construction, only the initial load and store annotation are propagated to the invoking object, where the final call site is concatenated to form the $cdu-1$. Note that the context level is inherent in the annotations, such that the same test tuple generation algorithm can be used for any context level with no modifications⁴.

⁴Within a call graph SCC due to recursion, call sites are concatenated to annotations as part of merge operations during iterative analysis according to context level (e.g., $cdu-3$ in previous paragraph).

VIII. SPACE/TIME COST ANALYSIS

To construct cdus for testing, space is required for the call graph of the CUT, as well as one *APE* graph for each method in the CUT. The call graph consists of one node for each method in the CUT and a set of edges for each call site, where the set of edges depends on the number of potential receiver objects at the call site. In the worst case, an *APE* graph is fully connected, that is, each object has possibly multiple references to every other object. In programs with long call chains and methods with multiple calls to the same method, it is possible to have an exponential number of annotations of exponential length on an *APE* graph edge with precise annotations. However, this effect can be avoided by limiting the length of annotations by selectively adding to annotations during the interprocedural merge, such as *cdu-1*, *cdu-2*, and *cdu-3*. When annotations are limited in this way, the time to construct the *APE* graph is proportional to the time to construct the points-to escape graph. In the worst case, the points-to escape graph construction algorithm by Whaley and Rinard takes $O(n^4)$ time where n is the number of *APE* graph nodes for the program[19].

The test tuple construction algorithm makes one pass over the call graphs representing the CUT. Because of the way the *APE* graph is constructed interprocedurally, it is adequate to make one pass through the nodes of a cycle in the call graph. This is due to the fact that the callee edges are merged into the caller's graph, and processing annotations during the merge does not alter any of the annotations in the callee's graph.

For each node in the call graph, the algorithm 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 the CUT. However, due to the marking scheme, and how edges are mapped during the merge of *APE* graphs at call

TABLE I
PROGRAM CHARACTERISTICS.

Name	Problem Domain	# of lines of user code	# of jvm instr		# of classes		# of methods		APE graph size		
			User	Lib	User	Lib	User	Lib	User	Lib	max size(Kb)
compress	text compression	910	2500	7070	17	90	50	301	6.1	1.1	43.0
db	database retrieval	1026	2516	11648	9	100	240	306	14.5	0.9	98.6
log	message log	300	125	16608	2	131	3	498	7.4	0.4	18.9
rich	task dispatcher	450	5807	3948	73	33	350	93	1.1	0.5	8.3
jasmin	java assembler	7500	19405	19469	56	137	258	582	3.9	1.5	247
jlex	scanner generator	7500	11000	7250	19	72	106	264	22.6	1.0	379
jess	expert system	9734	15200	13005	108	105	468	436	13.9	1.1	668
jack	parser generator	7500	22633	17547	59	149	317	552	1.3	0.4	133

sites, it is expected that the actual number of processed edges will be considerably less. For each processed *APE* graph edge, the algorithm examines each store annotation on the edge exactly once and each load annotation once for each store annotation on the edge.

Experimental Study of Space Costs: The major data structure for this testing approach is the *APE* graph. The *APE* graph construction algorithm has been implemented in the context of the FLEX compiler from MIT [13]. The experiments were run on a Sun Ultra 450 with 4 250MHz Ultra-II processors with 512 MB memory. Table I shows some general characteristics of the benchmark programs, including 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 that a relatively small program may rely heavily on libraries; therefore, analysis of the user program depends not only on the user code, but on the library code as well.

Table I shows the average storage requirements of the *APE* graph per method (user and library, columns 10 and 11), and the maximum *APE* graph storage requirement per benchmark (column 12). The storage requirements were calculated by computing the sum of two products. The first product is the total number of nodes over all the *APE* graphs

times the size of an *APE* graph node structure, and the second product is the total number of edges over all the *APE* graphs times the size of an edge structure.

As Table I shows, the average storage requirement per points-to escape graph is relatively small, which shows that the use of such a program representation for testing purposes is reasonable. The annotations are not included in this calculation of space, since we believe that our prototype does not reflect a fair assessment of storage for annotations (e.g., space could be reduced by partial or full annotations being reused; however, this would require a significant rewrite of our prototype implementation. The next table provides information on space for annotations.). From the results, the average size of graphs for library methods is consistently smaller than the size of the graphs for the user methods. This is because the graphs for the user’s code consist of a combination of the user and library graphs due to the interprocedural merge algorithm. An important consideration is that the compositional nature of the *APE* graphs avoids the requirement of keeping all *APE* graphs in memory at once.

The maximum *APE* graph represents the size needed to maintain the main method of the program. Essentially, the maximum *APE* graph contains subgraphs from all of its callees, which were merged into itself. Although *jlex* and *jack* have approximately the same number of lines of code, the average size of the program representation varies, 22.6 versus 1.3 Kb, respectively. The reason behind this discrepancy is likely due to the structure of the source code. For example, *jack* is designed in a more modular fashion with 59 versus *jlex*’s 19 user classes, and 317 versus *jlex*’s 106 user methods. *jack* has more user methods, most likely each small and creating a small number of objects, possibly causing each graph to be smaller than those constructed for *jlex*. Another reason for the disparity between *jack* and *jlex* could

TABLE II
ANNOTATION STORAGE REQUIREMENTS.

Name	cdu-1			cdu-2					cdu-3				
	ave number per edge		total cdus	ave length of annos		ave number per edge		total cdus	ave length of annos		ave number per edge		total cdus
	User	Lib		User	Lib	User	Lib		User	Lib	User	Lib	
compress	3	1.5	81	4.2	4.6	21	15	86	11.6	11.8	60	17	130
db	2.5	1.5	144	4.2	4.5	18.8	13.4	149	11.6	11.6	41	15	193
log	1.6	1.7	165	2.0	2.0	2.1	2.4	314	12.1	7.2	20.7	9.7	471
rich	1.9	1.7	175	3.5	3.7	7.4	8.9	175	13.6	8.4	61	6.2	343
jasmin	1.4	1.6	412	3.8	3.5	4.7	4.3	730	-	-	-	-	-
jlex	6.5	1.5	546	2.0	2.0	13.0	2.0	1320	-	-	-	-	-
jess	2.8	1.6	1593	-	-	-	-	-	-	-	-	-	-
jack	4.0	1.6	2006	-	-	-	-	-	-	-	-	-	-

be the differences in library usage. It is possible that the smaller size of points-to escape graphs of libraries for jack result in smaller size *APE* graphs for jack’s methods.

Table II provides insight into the storage requirements necessary for the edge annotations. The table shows the average length of each annotation for user and library code, as well as the average number of annotations per edge for user and library code for each type of cdu, and the total number of cdus. The average length of cdu-1 is not reported because it is always 2 (the invoking object and store (load)).

The table illustrates some interesting properties of cdus. First, the user and library annotations differ in length and number. This is due to the library annotations being merged into the annotations in user code graphs during the interprocedural merge. Next, cdu-1 is the most scalable in terms of the number of annotations per edge, but it also provides the least amount of context. However, cdu-1 still provides more context than a traditional def-use association. The annotation length is longer for cdu-2 and even longer for cdu-3. In addition, the number of annotations per edge increases substantially as the levels increase. For cdu-2 and cdu-3 for our larger programs, we observed that there exist many *APE* graph nodes with multiple outgoing edges with matching sets of annotations.

These edges are initially created due to the computed points-to information at the load statement; however, when this occurs in a method close to a call graph leaf, these same edges can be merged into multiple *APE* graphs. Without any reuse of annotation storage, our prototype runs out of memory during the building of the *APE* graph. These cases are reflected by blank entries in the table.

Although these results show that *cdu-3* is not scalable, it is likely that *cdu-3* could be useful for subregions of the code. Instead of constructing *cdu-3* annotations for the entire program, it is possible to construct *cdu-3* annotations for only certain critical regions.

From the same table, it is observed that the total number of computed *cdus* increases as the context level increases. The reason that more *cdus* are generated at each level is due to the generation of different call sequences that end in the same def-use association, creating multiple *cdus* for the same *cdu-0*. The numbers also provide evidence that the number of computed *cdus* is practical in that there is not an explosive increase.

IX. THE PROTOTYPE TESTING FRAMEWORK

The implementation of *cdus* is the basis of TATOO, an interactive testing environment to systematically test software. The prototype testing tool allows the tester to visualize the *APE* graph program representation, which characterizes how objects interact with other objects. In addition, the testing tool provides the tester with both visual and report-based coverage information about the CUT. TATOO also provides information about how objects interact with unanalyzed portions of code, as well as where objects may potentially escape through method calls or return statements. This information is useful in determining potentially fault-prone sections of code.

Figure 8 illustrates the two TATOO subsystems. The *program analysis* subsystem performs

the analysis to obtain the *APE* graph program representation. This subsystem is based on the extended FLEX compiler infrastructure [13]. In addition, a term representation used to view the *APE* graph, and an annotation table which maintains the information necessary for the testing subsystem are generated. The *testing* subsystem computes test tuples for testing the CUT by utilizing the annotation table. In addition, the testing subcomponent provides a graphical user interface that supports two primary features: test coverage identification and program representation visualization. The test coverage identifier, as shown in Figure 8(c), provides an environment that lets the user execute the program and then visualize coverage information. The tool consists of a code instrumenter and a trace analyzer, which provides coverage information about an executed program based on a given set of test tuples. The primary purpose of the program representation visualizer is to allow the user to visualize object interactions. We utilized the daVinci toolkit [6], which displays the program representation in a high quality manner, and facilitates communication with the GUI by allowing interaction between the *APE* graph and the source code, and testing information. A more detailed description of TATOO appears in [18].

X. SUMMARY AND FUTURE DIRECTIONS

This paper presented a strategy for constructing contextual def-use associations for structural testing of object-oriented software. By exploiting the combined points-to escape analysis developed for compiler optimization, the new testing paradigm does not require a whole program representation to be in memory simultaneously for testing analysis. Potential effects from outside the CUT are easily identified and reported to the tester. A prototype testing and analysis tool for object-oriented software, TATOO, was developed to demonstrate that the testing technique can easily be integrated into a testing framework.

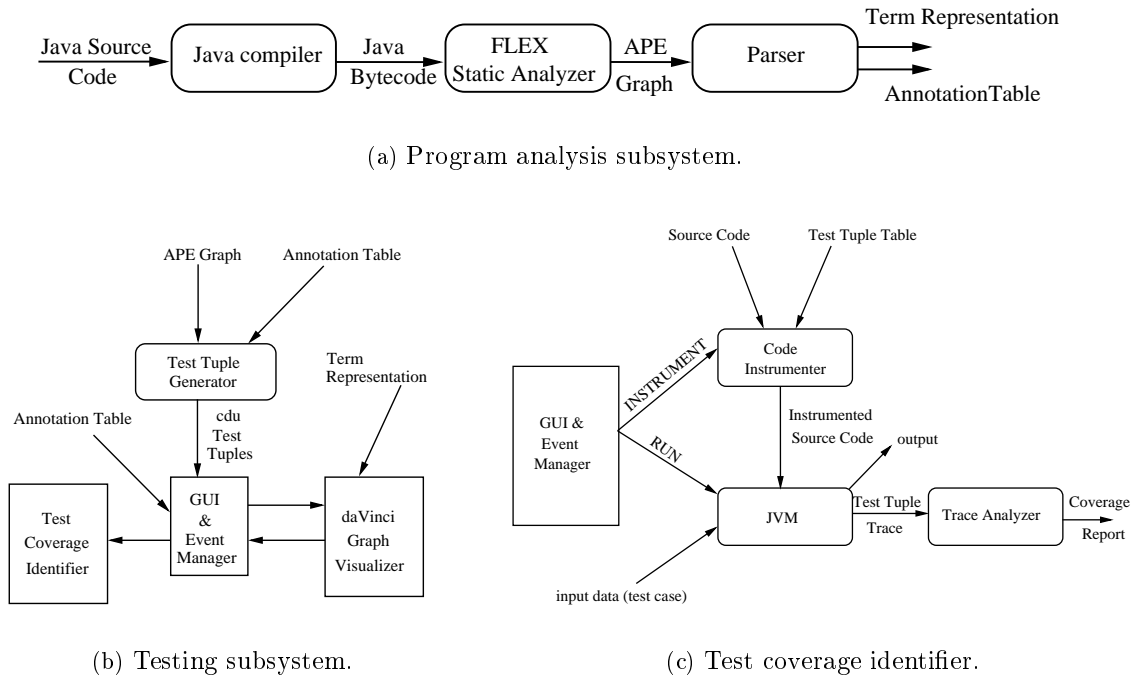


Fig. 8. TATOO subsystems.

In the future, we plan to investigate more scalable program representations and experimentally comparing fault detection capability of the method with different context levels. Our goal is to utilize the properties that the points-to escape graph exhibits, without the cost of the space overhead. In addition, we are looking at incorporating cdus into alternative programming paradigms, such as web-based programming. Finally, we are investigating the use of escape analysis for several software engineering tasks such as regression testing, priority-based testing, impact analysis, and program understanding.

REFERENCES

- [1] Roger Alexander and A. Jefferson Offutt. Analysis techniques for testing polymorphic relationships. In *TOOLS USA*, 1999.
- [2] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1996.
- [3] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated testing of classes. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, August 2000.

- [4] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1999.
- [5] Mei-Hwa Chen and Howard M. Kao. Testing object-oriented programs—An integrated approach. In *International Symposium on Software Reliability Engineering*, 1999.
- [6] M. Fröhlich and M. Werner. The graph visualization system daVinci. Technical report, Universität Bremen, Germany, September 1994.
- [7] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Proceedings of the Symposium on the Foundations of Software Engineering*, 1994.
- [8] Zhenyi Jin and A. Jefferson Offutt. Coupling-based criteria for integration testing. *Software Testing, Verification and Reliability*, 8(3):133–154, 1998.
- [9] Paul C. Jorgensen and Carl Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–8, September 1994.
- [10] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Chris Chen, Young-Si Kim, and Young-Kee Song. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–86, October 1995.
- [11] Alessandro Orso. *Integration Testing of Object-Oriented Software*. PhD thesis, Politecnico Di Milano, 1998.
- [12] J. Overbeck. *Integration Testing for Object-Oriented Software*. PhD thesis, Vienna University of Technology, 1994.
- [13] Martin Rinard. FLEX: The FLEX Group. <<http://www.flex-compiler.lcs.mit.edu>>. 2002.
- [14] Amie L. Souter. *Context-Driven Testing of Object-Oriented Systems*. PhD thesis, University of Delaware, 2002.
- [15] Amie L. Souter and Lori L. Pollock. OMEN: A strategy for testing object-oriented software. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, August 2000.
- [16] Amie L. Souter and Lori L. Pollock. Contextual def-use associations for object aggregation. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering*, 2001.
- [17] Amie L. Souter, Lori L. Pollock, and Dixie Hisley. Inter-class def-use analysis with partial class representations. In *Proceedings of the ACM Workshop on Program Analysis For Software Tools and Engineering*, 1999.
- [18] Amie L. Souter, Tiffany M. Wong, Stacey A. Shindo, and Lori L. Pollock. TATOO: Testing and analysis tool for object-oriented software. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 2001.
- [19] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.