Contextual Def-Use Associations for Object Aggregation [*]

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

ABSTRACT

This paper presents a novel formulation of definitions, uses, and def-use associations for objects in object-oriented programs by exploiting the relations that occur between classes and their instantiated objects due to aggregation. Contextual def-use associations are computed by generating a partial call sequence for each def and use based on object aggregation relations. By extending an escape points-to graph representation of the program, we have developed and implemented three strategies for achieving different levels of context for contextual def-use associations. Our experiments reveal that with all three strategies, multiple unique contextual def-use associations related to the same traditional (context-free) association are often generated. Contextual def-use associations are particularly useful for increasing test coverage and focusing the testing on critical method invocation sequences of object-oriented programs.

1. INTRODUCTION

Static def-use information has been shown to be useful not only for optimizing and parallelizing compilers, but also for debuggers, software testing, editors, program integration, and maintenance. In procedural languages, a *def* of a variable is an assignment of a value to the variable via a read or assignment operation, and a *use* of a variable is a reference to the variable, either in a predicate or a computation. A def-use association for variable `v` is an ordered tuple `(v, d, u)` where `d` is a statement where `v` is defined 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`.

In an object-oriented programming paradigm, not only is the computation of def-use associations complicated by polymorphism and inheritance, but object definitions and uses could be interpreted in a number of ways due to *object aggregation.* Programmers create object aggregation when they design a class that includes one or more objects of another class. Because the object-oriented paradigm promotes significant use of aggregation, manipulations to objects can have far reaching effects on the object definitions and uses of other objects through the aggregation has-a relation. From another perspective, an object C could be defined or used due to a definition or use of an object that is embedded in a possibly deep chain of aggregation has-a class relations with the object C.

In this paper, we present a novel formulation of individual object definitions and uses, which seeks to capture the object aggregation relation in addition to addressing inheritance and polymorphism. The resulting object def-use associations are *contextual* in that they provide context with the computed def-use associations in an object-oriented program, in contrast to traditional def-use associations, which we call context-free because they are reported free of any context. One useful application for *contextual def-use associations* is structural testing coverage that captures the object aggregation relations. Because a contextual definition (use) is generated as a partial call sequence ending in a given definition (use), multiple contexts for the same context-free association commonly result when object aggregation relations are captured. Because a single def-use association is considered to be covered by a test case that causes execution along any path that passes through the definition and later the use, 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, it is important to be able to select test cases that exercise the combined effects of different method invocations. Through contextual def-use associations based on object aggregation relations, we are focusing on critical combinations of method invocations. The impact is more thorough and focused testing to be performed for the manipulation of objects in object-oriented programs.

This new formulation was inspired by observations we made during our experimental studies of our OMEN approach to software testing [10] and our studies of the characteristics of Java programs [11]. In particular, by extending the escape points-to graph representation developed by Whaley and Rinard [12], we are able to generate contextual def-use associations with several strategies, each providing a different level of context, but all based on the program's object aggregation relations.

The main contributions of this paper beyond our previous papers are (1) a presentation of the def-use problem in the

[*]This work was supported in part by the National Science Foundation Grant EIA-98062525 and Grant EIA-9703088.

```
  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);  }

7      public Object pop(){
8         Object t = null;
9         if(isempty())
10           System.out.println("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; }  }


   class StackClient{
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;}

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

31    Node remove(){
32       Node e = this;
33       e = e.next;
34       return e; }  }
```

Figure 1: Object-oriented stack implementation.

presence of object aggregation with an example motivating contextual def-use associations and comparing them with context-free def-use associations computed by others (and our previous work), (2) development of a set of strategies for achieving different levels of context for contextual def-use associations, and (3) an experimental study comparing the different strategies for computing contextual def-use associations and traditional context-free def-use associations.

2. FORMULATION OF DEF-USE ASSOCIATIONS FOR OOP

The central role of an object in an object-oriented program makes it necessary to consider an object not as a single variable, but as a complex variable with state that can change through the manipulation of its attributes. Defining and computing def-use associations for objects is particularly complicated by (1) the impossibility of statically identifying the actual receiver of a message at some call sites due to polymorphism, and (2) the relations occurring between classes and their instantiated objects due to aggregation and inheritance. This paper focuses on the second issue, but the implementation of our techniques also addresses the first issue.

2.1 Example

The Java code in Figure 1 illustrates some of the issues involved in defining and computing def-use associations for object-oriented programs. For example, in class Node and class Stack, the set of the possible def-use associations includes: (data,25,27), (next,26,33), (top,13,16), and (top,6,3). Although these def-use associations are correct, they provide no context about how they are being used in conjunction with an object. For example, consider the def-use association, (next,26,33). In order to actually define or use the field next, a sequence of method calls associated with an object must be executed. The key observation is that if we use call sequences to characterize def-use associations of objects, we provide more context to the def-use associations.

Continuing with the example, we could calculate the following contextual def-use association of the object, s of type Stack: ((20-4-25), (22-12-27)). This notation describes that the object s is *defined* through a sequence of calls, (20-4) that lead to the field data being defined at line 25, which is associated with the class Stack through the aggregate relation with the Node class. Additionally, the object s is *used* after the execution of a sequence of calls that lead to the use of data at line 27, and there is a def-clear path between the def (20-4-25) and use (22-12-27).

2.2 Previous Work

Several researchers have presented definitions and algorithms for computing object def-use associations with the goal of addressing polymorphism and/or relations between classes[6, 1, 7, 11, 2]. In a prior paper[11], we presented a simple method for defining the def and use of an object in terms of state changes created by method calls potentially changing the values of the object's instance variables. The approach was based on computing flow-insensitive MOD and USE side effects. The computed def-use associations were not only context-free, but also lacked precision due to their flow-insensitive nature.

Alexander and Offutt developed an integration testing technique for object-oriented programs by extending their method coupling technique to handle inheritance, aggregation, and polymorphism[1]. Coupling-based testing requires that programs execute each definition of a variable in a caller to a call site, and then execute the uses of the corresponding formal argument in the callee. Similar to other definitions, they define an object to be defined when a value is assigned to a variable corresponding to an instance of a class. Additionally, they define an indirect definition of an object, *i-def*, to occur when a method m defines an instance variable bound to the object invoking m. Alternatively, an indirect use, *i-use*, occurs when m references the value of one of the object's instance variables. In their work, only one level of callees is considered for a def-use to be indicated (i.e., the def must occur in the callee for a coupling) and the reported def-use associations are context-free.

Chen and Kao[2] propose definitions for defining and using an object as follows: An object is defined if its state is initialized or changed, through one of the following acts (1) the constructor is called, (2) a data member is defined, or (3) a member function that modifies a data member is invoked. An object is used through the following conditions: (1) a data member is used in a computation or predicate, (2) a member function that uses the data member is invoked, or (3) the object is passed as a parameter in a function call.

After computing intramethod def-use sets by traditional data flow analysis techniques, they build an object control flow graph (OCFG) to compute the interprocedural object def-use pairs. Each method of the class is represented as a supernode containing a CFG for its internal control flow. Call graph edges are represented as message-passing edges from the calling method's call site node in its CFG to the supernode of the callee method. Method-def-use edges are inserted between supernodes based on intramethod def-use pairs and message-passing edges in order to direct further propagation. An iterative algorithm adds method-def-use edges and intermethod def-use pairs until the OCFG stabilizes. The object def-use pairs computed by Chen and Kao fall into the category of intermethod def-use pairs as defined by Harrold[4], which are interprocedural def-use pairs that exist regardless of client usage of the class being tested. Chen and Kao suggest that interclass object def-use pairs can be computed using interprocedural data flow techniques developed by Harrold and Soffa[3]. However, these techniques do not address the complications of object-oriented features, in particular, aggregation.

The work by Chen and Kao is the most closely related to our work. Unfortunately, several key descriptions and assumptions of their technique are left unclear in their paper. In particular, it is not clear whether the computation of method def-use sets handles multiple(repeated) composition of objects. Without this capability, Chen and Kao do not fully address the potential aggregation of objects, where a data member may be an object rather than a primitive type. The terms intraclass and interclass are not well defined and do not appear to be consistent with definitions by other authors[4, 11]. Most importantly, the computed def-use pairs are context-free.

3. DEFINING CONTEXTUAL DEF-USE ASSOCIATIONS

We define a contextual def-use association to be a def-use association in which the def and use contain context beyond the variable and the location of the definition and use. Our work focuses on providing context to defs and uses to reflect the object aggregation relations. An aggregation relationship is also known as a *has-a* relationship. UML terminology defines a stronger form of aggregation called composition, where the aggregate(whole) object is responsible for memory allocation/deallocation of the parts that compose the whole object. We do not distinguish between different forms of whole-part relationship between objects. In this paper, aggregation is defined as follows: a relationship between classes where one class contains an object of a second class[9]. An example of an aggregate relation is shown in Figure 1, where class Stack contains an instance of class Node as one of its parts.

In our case, context is a sequence of method call sites ending with the location of the actual load (setting a reference to point to the object referenced by an object field, $r_1 = r_2.f$) or store (setting a field of an object to point to an object, $r_1.f = r_2$) of an object field. The call sequence may or may not be a complete call sequence, but rather a partial call sequence that does not include every call site that needs to be executed to lead to the actual load or store of an object field.

We have defined four levels of context for computing contextual def-use associations based on object aggregation. We use the naming convention **cdu-x** to indicate the strategy that computes contextual def-use associations with context level **x**. Higher context levels provide more context with the def and use, but with the tradeoff of increased space (and possibly compute time) requirements.

cdu-0: The base case is traditional, context-free def-use associations, of the form *(v, def, use)* where *def* is a statement defining a variable v that is potentially used at statement *use*.

cdu-1: Each *cdu-1* is a tuple *(o,def,use)* for an object o in which *def* (*use*) is defined to be a pair $(CS_{om}\text{-}L)$[1]. CS_{om} is the call site of the method call leading to a modification of the state of object o. L is the location of the actual store (load) causing the modified state for object o.

cdu-2: Each *cdu-2* is a tuple *(o,def,use)* for an object o in which *def* (use) is defined to be a sequence of the form $(CS_{om}\text{-}(CS_{scc_{entry}}\text{-}CS_{scc_{exit}})^{*}\text{-}L)$. The first and last entries of the sequence are the same as *cdu-1*. The internal sequence is a sequence of pairs where each pair consists of the entry and exit nodes of a strongly connected component in the call graph. This context level represents a def (use) of object o as a partial static call sequence from a call to object o's method including only the entry and exit of each strongly connected component along the call chain to the store (load) causing the object state change. The sequence is partial because there may be several calls between the call sites included in the context which are not represented, and thus there may be multiple call sequence paths associated with any two consecutive call sites represented in the contextual def-use association.

cdu-3: Each *cdu-3* is a tuple *(o,def,use)* for an object o in

[1]We represent the tuples for defs and uses with hyphens to distinguish them from the def-use associations which are represented as tuples.

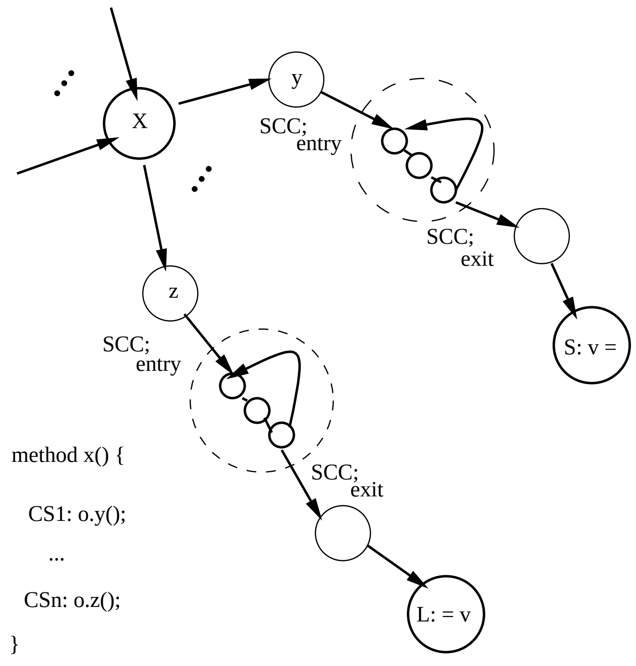

Figure 2: Illustration of contextual du.

which *def* is defined to be a sequence (CS_{om}-CS_1-...-CS_m-L) where the first and last entries of the sequence are the same as *cdu-1*, and the internal sequence of CS_i are call sites in a call sequence leading from the original call site to the store, with each strongly connected component (SCC) represented by a single sequence of call sites through the SCC (i.e., no multiple passes through the SCC are included). This context level represents the object aggregation relations more completely than the lower context levels, but at the expense of increased space requirements.

Figure 2 shows a portion of a call graph used to illustrate *cdu* forms. Consider the *cdu-1* for *cdu-0*, (V, S, L), that starts at node x and ends at the nodes containing the def and use of v, which we can assume is a field associated through an aggregation relationship of object o. The *cdu-1* is represented as $(o, CS1 - S, CSn - L)$. To provide more context to the def and use, we use *cdu-2* and define the association as follows: $(o, CS1 - SCC; entry - SCC; exit - S, CSn - SCC; entry - SCC; exit - L)$. Finally, to provide even more context to the load and store of object o, we use *cdu-3*: $(o, CS1 - ... - SCCentry - ... - SCCexit - ... - S, CSn - ... - SCCentry - ... - SCCexit - ... - L)$, where the ... would contain a sequence of nodes in the SCC.

4. CONSTRUCTING CONTEXTUAL DEF-USE ASSOCIATIONS

To compute contextual def-use associations, we extended and modified the points-to escape graph program representation developed by Whaley and Rinard [12]. 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. 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.

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. There are two distinct kinds of nodes, namely, inside and outside nodes. An inside node represents an object creation site for objects created and reached by references created inside the current analysis region of the program. In contrast, an outside node represents 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. The distinction between inside and outside nodes is important because it is used to characterize nodes as either captured or escaped. The escape information is particularly useful for analyzing incomplete programs and for providing feedback to a software tester.

We have extended the points-to escape graph by adding annotations to edges in the graph. The annotations provide information about where basic object manipulations i.e., loads and stores of objects, occur within a program. Our resulting representation is called an *A*nnotated *P*oints-to *E*scape (ape) graph. A detailed description of the ape graph representation, construction algorithm, and examples are presented in [10]. Here, we give a brief overview.

Figure 3 shows an example set of annotations on a single edge of an ape graph. The edge labeled next represents a reference from a field named next of the object of type Node, annotated with both a load and store annotation. The annotations indicate that there exist both a load and store of the field next. Further, the location where the load and store occurs is maintained through the annotations. The annotation, (store 20-4-26), represents two calls, one invoked on line 20 of the program. The second call invoked on line 4 can lead to a store of an object into the field next at line 26. Similarly, the load of the field next occurs at line 33, following a chain of calls from lines 22 and 13.

Before an ape graph representation is constructed, a precise call graph is created. Then, we build one ape graph per method in the program. For a given method, an initial ape graph is constructed to represent the parameters and class objects on entry to the method. The ape graph is refined by repeatedly processing the object-related statements in the CFG of the method until a fixed point is reached. At each statement, ape graphs representing the predecessors of the statement in the CFG are first merged into a single graph. The new ape 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

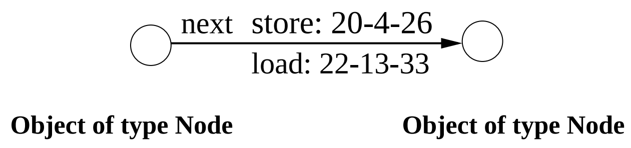

Figure 3: Illustration of ape graph annotation.

Name	Problem Domain	# of bytecode instr		# of classes		# of methods	
		User	*Library*	*User*	*Library*	*User*	*Library*
log	message log	125	16740	2	131	3	500
jload	java installer	322	17235	1	153	3	613
echo	debugging web server	342	18571	3	153	14	528
compress	text compression	2500	7070	17	90	50	301
db	database retrieval	2516	11648	9	100	240	306
richards	task queues	5807	3948	73	33	350	93

Table 1: Benchmark program characteristics.

the analysis of a 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.

Interprocedural analysis is achieved through merging the parameterized ape graphs of the potentially invoked methods at a call site with the ape graph at the point immediately before the call site, to form the ape 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 call graph.

A given load/store annotation on an ape graph edge is incrementally computed by modifying the points-to escape graph merge algorithm performed at a call site to merge a callee's ape graph into a caller's ape graph. As an edge is mapped from a callee's ape graph into the caller's ape graph at a call site in the caller, the statement number, and the earliest visible statement for the call site are concatenated onto the annotation sequences mapped from the callee's ape graph.

The contextual def-use associations are computed by a separate topological traversal of the call graph starting at the root, analyzing each call graph node's ape graph. Duplicate contextual def-use associations are avoided through a marking scheme. The load and store annotations on each ape graph edge are analyzed to identify contextual def-use associations. Different context levels for contextual def-use associations can be obtained by altering the way that ape graph edges are annotated, without altering the actual def-use association computation pass. Since annotations are incrementally augmented during the operation of merging a callee's ape graph information into a caller's ape graph, the different strategies are all implemented by different versions of the merge operation. The versions differ in which call sites are actually saved as part of an annotation as it is being incrementally constructed.

5. EMPIRICAL STUDY

We have implemented the computation of contextual def-use associations within the FLEX compiler framework from MIT [5]. In particular, we have extended the points-to escape graph construction within the FLEX compiler to create an ape graph program representation, which provides the environment to build the different levels of contextual def-use associations. We have conducted experiments on several Java benchmarks to compare the total number of contextual def-use associations (cdus) computed by each context level strategy, as well as the distribution of different numbers of cdus computed for the same context-free def-use association (du) under each context level strategy.

Name	cdu-0	cdu-1	cdu-2	cdu-3
log	67	165	314	471
jload	82	173	515	716
echo	97	162	283	356
compress	44	81	86	130
db	62	144	149	193
richards	130	175	175	343

Table 2: Total number of cdus per context level.

Table 1 provides insight into the nature of the benchmarks we have used for our experiments. The general characteristics of a program have been separated into the user and library components of the program, where library refers to classes in the Java class library. The columns in Table 1 show the number of (user and library) JVM instructions, classes, and methods in each benchmark. We analyzed the entire program including the parts of the Java library used, which makes the benchmarks overall substantially larger than the user code alone.

Table 2 presents the total number of computed cdus by each context level strategy for each benchmark. Figure 4 illustrates the percent increase in the number of cdus over the number computed by cdu-0. From both of these figures, we observe that as the context level increases, the number of computed cdus increases. In all cases, context level 3 results in a substantially higher percent increase. We believe that the larger number of cdus for the programs log and jload are attributed to the heavy use of object-oriented design. Even though these benchmarks have a small number of user classes, we noticed that jload and log have more complex class relationships with the library classes than the other benchmarks.

Figure 5 shows the cumulative distribution of the number of unique computed cdus per context-free du (cdu-0). For example, the cumulative distribution over all the benchmarks for cdu-1 reflects 50 instances where two unique cdus were reported in place of a single context-free du. Looking at the data for the 1-1 ratio of cdus to context-free cdus, we observe that as the context of the def-use pairs increases (cdu-1 to cdu-3), the frequency of the 1-1 def-use pairs decreases. As the context level increases, more cdus are distributed over the set of other ratios higher than one. However, most of the ratios that we observed were between 2-1 and 10-1. These results suggest that testing coverage could be increased by contextual def-use associations, but the number of additional def-use associations does not explode to become impractical.

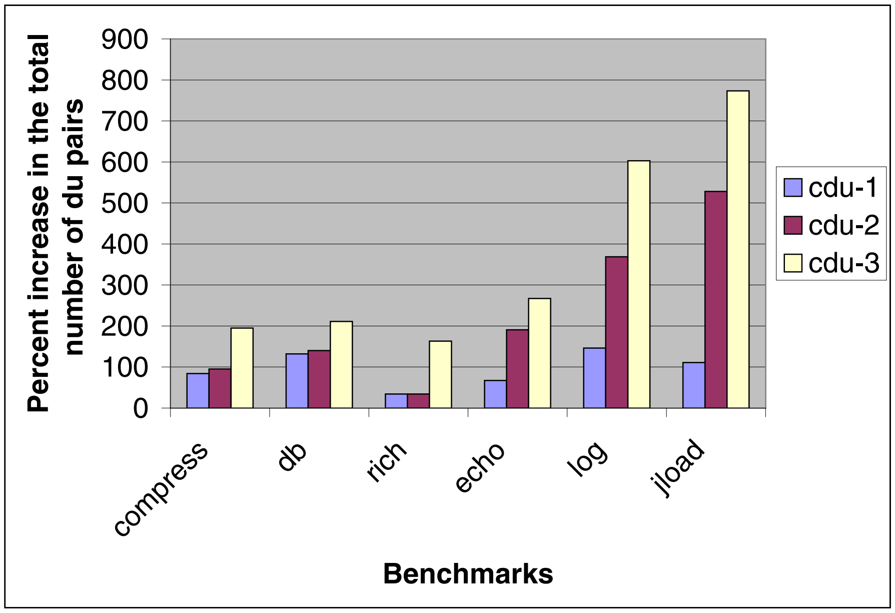

Figure 4: Percent increase in number of cdus over cdu-0.

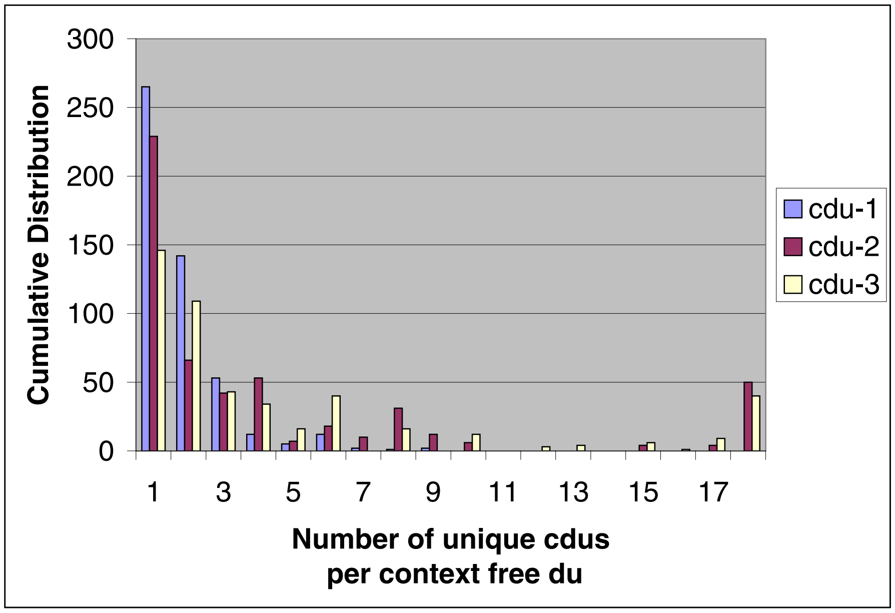

Figure 5: Cumulative distribution of unique cdus per context-free du.

6. APPLICATION TO SOFTWARE TESTING

Traditional data flow testing uses coverage criteria to select subpaths in the program for testing based on sets of du chains[8]. After the du chains have been computed, test cases are generated, manually or automatically, to exercise the du chains according to a selected coverage criterion, such as all-defs, all-uses, and all-du-paths[8].

However, using traditional data flow testing coverage criteria for object-oriented programs may fall short on achieving good coverage when aggregation relationships need to be thoroughly tested. For example, consider using the all-defs criteria, which requires that for each definition a path to at least one reachable use is exercised. A simple path that covers the def and one use of a field will satisfy this coverage requirement. But several different objects could be associated with this one field and the context in which they are defining and using the field will not be taken into consideration when using this criterion. If we try to strengthen the criterion by using the all-uses criterion, this criterion requires that for each definition, paths to all reachable uses are exercised. In this case, we have no means of determining the different contexts associated with the definition of the field. In other words, there is no association with the object whose state changes due to the definition of one of its fields. By providing different levels of context through cdus, we are able to provide better coverage in terms of objects and their related fields.

7. CONCLUSIONS AND FUTURE WORK

The main contribution of this paper is the introduction of the concept of contextual def-use associations. Our definition of contextual is based on the object aggregation relation. We show how different context levels can be achieved with minor changes to the underlying program representation and algorithms for computing contextual def-use associations. One major application for this work is in software testing — both for increasing test coverage and focusing on critical combinations of method invocations.

We are currently trying to run larger programs through our implementation in order to better understand the characteristics and implications of contextual def-use chains. We are also investigating techniques to remove cdus that are currently included, but can be shown to be infeasible based on analyzing more precise type information. Lastly, we are investigating other possible applications of contextual def-use associations.

8. REFERENCES

[1] Roger Alexander and A. Jefferson Offutt. Analysis techniques for testing polymorphic relationships. In *TOOLS USA*, 1999.

[2] Mei-Hwa Chen and Howard M. Kao. Testing object-oriented programs - an integrated approach. In *International Symposium on Software Reliability Engineering*, 1999.

[3] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[4] M.J. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. In *Proceedings of the Symposium on the Foundations of Software Engineering*, 1994.

[5] M. Rinard et. al. FLEX. www.flex-compiler.lcs.mit.edu, 2000.

[6] John D. McGregor, Brian A. Malloy, and Rebecca L. Siegmund. A Comprehensible Program Representation of Object-Oriented Software. *Annals of Software Engineering*, 1996.

[7] A. Orso. *Integration Testing of Object-Oriented Software*. PhD thesis, Politecnico Di Milano, 1999.

[8] S. Rapps and E. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[9] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[10] A. Souter and L. Pollock. OMEN: A Strategy for Testing Object-Oriented Software. In *Proceedings of the International Symposium on Software Testing and Analysis*, August 2000.

[11] A. Souter, 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*, September 1999.

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