

Inter-class Def-Use Analysis with Partial Class Representations *

Amie L. Souter and Lori L. Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
{souter, pollock}@cis.udel.edu
(302) 831-1953

Dixie Hisley
HPC Division
U.S. Army Research Lab
APG, MD 21005
hisley@arl.mil
(410) 278-9156

Abstract

Object-oriented program design promotes the reuse of code not only through inheritance and polymorphism, but also through building server classes which can be used by many different client classes. Research on static analysis of object-oriented software has focused on addressing the new features of classes, inheritance, polymorphism, and dynamic binding. This paper demonstrates how exploiting the nature of object-oriented design principles can enable development of scalable static analyses. We present an algorithm for computing def-use information for a single class's manipulation of objects of other classes, which requires that only partial representations of server classes be constructed. This information is useful for data flow testing and debugging.

1 Introduction

The use of object-oriented program design principles has created a new set of problems for software tool developers and compiler writers. Besides having to represent and analyze programs that exploit the new features of classes, inheritance, polymorphism and dynamic binding, the style used to develop software applications and the units of analysis are changed. Applications consist of a set of classes (with a main program or main class, depending on the language), which contain calls to methods within these classes as well as calls to methods in classes written by others. A library consists of one or more classes without a main program. A programmer may design and implement just a single class or library as their designated programming goal, without a specific targeted application which will use the class. Thus, a class becomes a natural unit of programming, compilation, debugging, and testing. However, the context of the class is often unknown. Thus, the order in which methods of the class will be called from client applications is unknown.

*This work was partially supported by NSF under grant EIA-9806525.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PASTE '99 9/99 Toulouse, France
© 1999 ACM 1-58113-137-2/99/0008...\$5.00

Each class is composed of instance variables and a set of methods which must be called in order to manipulate those variables. Each time a class is instantiated, a new object of that class is created with its own local instance variables. The state or value of the object's instance variables is changed only through method calls to that object; the value (state) is retained between method calls to that object. A class may also have class variables which are shared among different instantiated objects of that class. These variables can be changed by a method call to any object of that class.

A class C can be used as the type of a local variable declaration in a method of any class (including class C), or as the type of an instance variable or class variable of any class (including class C). Objects of class C are not instantiated until a call to class C's constructor is made. Often, these instantiations of class C occur in other classes which are using class C. Therefore, in addition to inheritance as a mechanism for code reuse, code reuse is common in the building of classes as common *server* classes, which then may have multiple *client* classes which instantiate objects of that class and call methods of that class. A class may serve in the role of *client* class of another class, as well as a *server* class for other classes. Typically, the server and client classes are not written at the same time or by the same programmer.

Object-oriented design commonly results in many methods, each with a small number of statements and simple intraprocedural control flow. McGregor et al [13] identify two specific kinds of methods: modifier and read-only. A modifier method defines or redefines instance variables in a class (and possibly also references instance variables), while read-only methods only reference instance variables; no changes to instance variables occur.

Static def-use information has been shown to be useful not only for optimizing and parallelizing compilers[14], but also for debuggers[18], software testing[8], editors[16], program integration[9], and software maintenance [6]. 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 pair for variable *v* is an ordered pair (*d*, *u*) where *d* is a statement where *v* is defined and *u* is a statement that is reachable by some path from *d*, and uses *v* or a memory location bound to *v*. McGregor et al [13] describe the *def* of an object as occurring when an object is sent a modifier message. The *use* of an object occurs when

```

Class Stack{
    int myStack[];
    int top;

1  public Stack(int s){      myStack = new int[s];
2                          top = 0;      }
3  public void push(int obj){ myStack[top++] = obj; }
4  public int pop(){
5                          if(size() == 0) return error;
6                          int temp = myStack[top-1];
7                          top--;
8                          return temp;    }
9  public int size()        { return top;    }
10 public int top()         { return myStack[top-1]; }
11 public boolean isEmpty() { return top == 0; }
12 public void example()    { push(10);
                          if(size() == 5)
                              do_something++; }
}

```

Figure 1: Illustrating kinds of def-use pairs.

an object is sent a read-only message or when the object appears as an actual parameter to a message. We elaborate on these definitions as part of this paper.

Due to code reuse benefits of building common *server* classes used by multiple *client* classes, the computation of def-use information for a single class will almost certainly involve tracking def's and use's of objects of other classes. Existing def-use analysis techniques for object-oriented programs [3, 7, 10, 11] do not explicitly address the problem of def-use analysis of variables that are instantiated objects of other classes.

In this paper, we show that existing techniques for def-use analysis of a class can be extended for this problem, but require building a graph representation for the whole set of interacting classes, before the analysis is performed. When one considers the potential interactions between classes created through this client-server class relationship, the size of the program representation needed for def-use analysis of objects involved in analyzing a single class can become very large. Based on our study of a set of real Java benchmark classes and applications obtained from various sources, we show that much of this representation is unnecessarily constructed for analyzing a single class. We present an approach for computing def-use information for a single class's manipulation of objects of other classes, which requires that only partial representations of server classes be constructed.

2 Def-use Analysis of Classes

2.1 Definitions

In data flow analysis of classes, there are several different kinds of def-use pairs in a class depending on the location of the instance variable definitions and references: intra-method, inter-method, and intra-class [7]. To illustrate the different kinds of def-use pairs, we use the Stack class defined in Figure 1. Intra-method def-use pairs have the same meaning as they do in a procedural language; a definition of a variable and a subsequent use are both located in the same method. An example is local variable *temp* defined at line 6 and used at line 8 of class Stack's *pop* method. Techniques

for computing intraprocedural data flow analysis can be used for intra-method data flow analysis of primitive types [14]. Inter-method def-use pairs have the same meaning as inter-procedural def-use pairs have in a procedural language. In class Stack, the method example illustrates an inter-method def-use pair (3,9) for the variable *top*. The call to *push* defines *top* and the call to *size* uses *top*. Inter-method def-use pairs result from the calling relations between methods of a class disregarding the unknown sequence of calls after instantiation. Intra-class def-use pairs arise due to sequences of method invocations that arise if the class was instantiated. An instantiated class can call methods in any order. For example, the method call sequence *push*, *pop* generates def-use pair (3,6). The instance variable *top* is defined in *push* and used in *pop*.

Due to code reuse through client and server classes, detection of each of these kinds of def-use pairs can involve inter-class analysis. Def-use pairs that involve inter-class analysis are illustrated using the *genericStack* and *Array* classes in Figure 2. The figure only shows the seven methods of *Array* used by *genericStack*, while in fact, *Array* has 59 methods in all. The *genericStack* class has an instance variable *myStack* of type *Array*. Methods defined in *genericStack* manipulate the instance variable *myStack* through calls to methods defined in the *Array* class. Therefore, def-use pairs in the *genericStack* class involve an object of the *Array* class. For example, the method call sequence *push*, *top* in *genericStack* has associated with it def-use pair (9,3) from the *Array* class. The instance variable *myStack* of class *genericStack* is redefined in terms of its instance variables *myLength* and *myStorage* in the method *add*. The method *back* uses the instance variable *myLength* associated with *myStack*.

2.2 Def-use Analysis Techniques and Related Work

Previous research on data flow analysis of classes has focused on intra-method, inter-method, and intra-class def-use pairs of primitive types [7]. They define the relationship between these def-use pairs and provide program representations to generate the def-use pairs. These program representations are the class call graph, a frame around the class call graph, and the Class Control Flow Graph (CCFG). The frame allows for a random calling sequence between methods in a class, and enables techniques for interprocedural def-use analysis to be applied to detect def-use pairs of primitive types in the class with different levels of precision due to aliasing effects and the techniques used for dealing with aliases[8, 15].

In later work, Larsen and Harrold[10] introduce the concept of a class dependence graph and adapt the system dependence graph for use in object-oriented software. Their program representation fully represents the features of object-oriented software including inheritance and polymorphism. They use this program representation for slicing object-oriented software[10]. Liang and Harrold extend the program representation once more for use with object slicing[11]. One benefit of this program representation is its ability to be used with existing slicing algorithms. Another benefit comes from the large amount of information that is stored in the graph, making it useful for a large number of software engineering applications. A drawback of the program representation

```

Class genericStack{
    Array myStack;

1 public genericStack() { myStack = new Array(); }
2 public Object top() { return myStack.back(); }
3 public void push(Object o){ myStack.pushBack(o); }
4 public Object pop() { return myStack.popBack(); }
5 public bool isEmpty() { return myStack.isEmpty(); }
6 public int size() { return myStack.size(); }
}

Class Array {
    Object myStorage[];
    int myLength;

1 public Array() { myStorage = new Object[SIZE];
2 myLength = 0; }
3 public Object back() { return myStorage[myLength-1];}
4 public Object popBack() { return myStorage[--myLength];}
5 public void pushBack(Object o){ add(o); }
6 public Object front(Object o){ return myStorage[0]; }
7 public bool isEmpty() { return (myLength == 0) }
8 public int size() { return myLength; }
9 public void add(Object o){ myStorage[myLength++] = o; }

//Array has 46 other methods
}

```

Figure 2: Def-use pairs involving inter-class interactions.

is its size. Their work is used to represent single classes, incomplete systems, as well as complete systems. When analyzing systems with a large number of unrelated classes, the program representation grows very large with the use of many interacting classes.

McGregor et al [13] developed the object-oriented program dependence graph (OPDG) for investigating object-oriented software. The program representation they introduce is broken into three layers, representing the class hierarchy relationship among classes, control and data dependence relationships, and the runtime information about object-oriented programs. Their work introduces the terms *object flow* and *object dependence* of object-oriented software. Object flow is described as how an object's state can change from one program point to another through changing the values of its instance variables. The term object dependence is used to describe the interaction of objects in a program. Dependence is established through message sends, message receives, parameter passing of objects, control dependence on an object, and shared attributes of an object. They illustrate two uses of their representation, namely, slicing and a simple class metric. This program representation can be used to represent a single class or a complete system. A drawback of their approach is scalability for representing large programs. Although their approach is modular in the sense that it is created layer by layer, the first layer and second layer must be built in order to build the third layer. Also, the size of the graph will grow large as the number of classes introduced into the program increases.

Recently, scalable analysis of object-oriented software has become a focus of several research efforts. Tip et al [17] describe an algorithm for slicing class hierarchies in C++ programs. One of their motivating factors for slicing class hierarchies is to decrease space requirements by eliminating

unreachable code. An example of such code includes portions of the class hierarchies which are not accessed, therefore they are not needed to run the program and do not need to be included in the executable version. This is a common situation when libraries are used.

Chatterjee, Ryder, and Landi introduced relevant context inference (RCI) as a modular technique for flow and context-sensitive data flow analysis of statically typed object-oriented programming languages[4]. Modularity makes it possible to avoid having the entire program representation in memory at the same time. Their approach is scalable in terms of memory requirements. Their approach is capable of analyzing complete systems as well as incomplete systems such as libraries. Most recently, RCI has been used in the application of data flow based testing of object-oriented libraries[3]. Their technique focuses on the difficulties attributed to testing libraries due to unknown alias relationships between parameters, unknown concrete types of parameters, dynamic dispatches, and exceptions. These difficulties arise from the fact that no driver program exists in library testing. In their paper, they present an algorithm for finding def-use associations in object-oriented libraries, which addresses these issues.

In summary, none of these efforts have explicitly addressed the def-use analysis of objects of server classes.

3 Inter-Class Def-Use Analysis

We first present an approach to computing def-use pairs for a single class's manipulation of objects of other classes using a server-based approach, and then a second approach which takes a client-based approach. The client-based approach exploits the characteristics of object-oriented program design to reduce the amount of program representation constructed and analysis performed. In each algorithm, we describe how to deal with a single server class of the client; the algorithm would be executed for each server class used by the client.

Let class C be the client class for which we are computing def-use pairs for an instantiated object of another server class S. We assume that the class control flow graph (CCFG) is built for C, in order to compute def-use pairs of primitive types within C [7]. By adding the special frame of nodes and edges to the class call graph prior to building and connecting the individual control flow graphs for each method of C, the CCFG represents possible flow of control between methods due to arbitrary sequences of calls to public methods.

The first step in both algorithms is to build a call graph representing the potential control flow between methods of server class S, resulting from method calls in C. Any one of a number of available techniques can be used to construct the call graph, each leading to a different cost and level of precision[5, 1]. As long as the resulting call graph is a safe overestimate of the actual call graph, the results of def-use analysis will be safe. A more precise call graph will have less edges, and thus the def-use analysis may be more efficient and more precise. Inheritance and polymorphism can cause the call graph of S to contain methods from superclasses or

Client Class	Server Class	Total # Server Methods	Total # of CallGraph Methods	% Used by Client	Total # of CallGraph Modifiers	Ave # of Branches per Method	Total # of CFG nodes in Subgraph	Total # of CFG nodes in Server
Stack	Array	59	21	35	8	0.29	238	1259
ICount	ClassInfo	18	4	22	2	2.50	245	459
	Routine	46	2	4	1	1.00	28	1664
Map	BasicBlock	15	2	13	1	0.50	22	350
	RBtree	31	31	100	14	2.77	1247	1483
SplayTree	RBtreeIt	11	2	18	2	0.0	38	106
	BSTree	39	7	18	7	0.43	227	706
P.Matcher1	BTreeNode	41	15	37	9	0.60	93	919
	WcardMatcher	7	5	71	1	5.4	409	487
Shuffling1	MalPatExc	2	1	50	1	0.0	4	7
	ElementNode	13	6	46	3	0.8	23	189
Stats	P.Matcher	5	1	20	1	1.0	3	10
	Swapping	4	1	25	1	0.5	11	53
Sequence2a	Array	59	4	7	1	0.9	70	1259
	Shuffling	6	1	17	1	0.7	54	87
Jtu	TrekException	2	2	100	1	0.5	37	37
	Trek	75	10	13	4	3.0	478	2694
DLList	DLList	38	1	3	1	1.2	21	1034
	Sequence	28	2	7	1	0.3	3	221
InputIt	Container	50	10	16	1	0.14	42	442
	NonMutating	14	1	7	1	2.1	18	384
DLListItem	DLListItem	3	1	33	1	0.3	6	43

Table 1: Characteristics of client-server class usage.

subclasses of S. In this paper, we refer to these methods as if they were part of S for ease of presentation.

While a call graph for an application will have a single *entry* node representing the main program, this call graph will have multiple *entry* nodes, one for each method in class C that contains call sites to methods in class S. Edges in the call graph are either edges from methods in C or between methods in S with a calling relationship. Note that the resulting call graph for server class S need not be the complete call graph for all of class S. It needs to contain nodes only for methods potentially called directly or indirectly through call sites in C. Thus, we refer to this call graph as S's call subgraph with respect to C. Section 3.2 presents empirical data comparing the size of this constructed call subgraph to the size of the complete call graph for S.

3.1 Server-based Algorithm

If we view the def (use) of an object as a def (use) of one of its instance variables, then we can view the problem of computing def-use pairs for manipulation of objects of S within C as computing the def-use pairs of instance variables of S with respect to S's use by class C only. After building the conservative call subgraph for S, the second step is to build the CCFG for S's call subgraph (excluding the frame because C becomes the driver of the order of calls to S's methods). The third step is to compute all def-use pairs of instance variables of S with respect to calling sequences within C, using Harrold and Rothermel's techniques [7].

With this approach, def-use analysis of a client class C's use of objects of another class S computes def-use pairs of the server class's instance variables with respect to potential calling sequences from client class C. If S's methods use objects of other server classes, then the representation and def-use analysis continues to grow in time and space requirements. However, this approach requires the CCFG representation for both C and for the call subgraph of S rooted in C's methods, and any other server classes needed for S and its subsequent server classes. In section 3.2, we show that this kind of analysis becomes impractical when one examines the extent of client-server class relations resulting from object-oriented design.

As part of our research efforts, we have reexamined the concept of def-use pairs in the context of how they are often applied. For example, for data flow testing, this server-based approach defines data flow testing of class C as testing all intra-method, inter-method, and intra-class def-use pairs of primitive types within C, in addition to all def-use pairs of instance variables of S, created through method calls in C to S's methods, and other server classes due to S and its server classes. The following section takes a closer look at the characteristics of client-server classes to provide more insight into how much of this information is actually necessary to adequately analyze def-use relations within C.

Name	Description	Source	# of classes
class_inter	bytecode analyzer	BIT: University of Colorado	51
javacc	parser generator	Sun	57
jjtree	preprocessor for JavaCC	Sun	61
jjdoc	documentation generator for JavaCC spec	Sun	43
jd	graphical debugger	IBM	41
jimple	translator: bytecode to three address code	McGill University	303
sableCC	object-oriented compiler-compiler	McGill University	234
members	report generator showing members of a class	Jtrek: Digital/Compaq	25

Table 2: Java program descriptions.

3.2 Empirical Observations

We analyzed a set of real Java classes from various sources¹ to observe the use of object-oriented design principles. The individual classes are sizable, ranging from 2-75 methods in an individual class.

Table 1 presents some of the interesting characteristics discovered in the set of benchmarks. The first column shows the total number of methods in the server class. The second column shows the number of methods of the server class which are actually represented in the call subgraph of the server rooted in calls by the client class. The third column uses these two columns to present the percent of the methods of the server class that actually show up in the call subgraph. The fourth column shows that an even smaller portion of the server methods in the call subgraph are actually modifiers, in the sense that they lead to actual changes in the state of the object through definitions of instance variables of the server. Column 5 reports the average number of control flow branches per method in the server class. Finally, column 6 presents the total number of control flow graph (CFG) nodes in the CCFG for the server class subgraph and column 7 shows the total number of CFG nodes needed to represent the entire CCFG for the server class.

This table shows that on average a small percentage of methods defined in the server class are actually used by the client class. This observation is not a surprise, but indicates that the call subgraph that is built is much smaller than the complete call graph of the server class. The difference between the last column and the prior column of Table 1 indicates the amount of space savings in terms of the number of nodes in the CCFG when only nodes in the call subgraph are considered. Second, of those methods called by the client class, approximately only half are modifier methods. These are the only methods that could lead to changes in the state of the object being used within the client. Third, the branch count within a method is often zero, and on average between 0 and 3.

We also analyzed a set of real Java programs which utilize from 25 to 300 server classes in order to obtain a characterization of larger programs involving many classes. To demonstrate the complexity of class interactions in these

large programs and the need for scalable program analysis, Figure 3 illustrates a class interaction diagram, which documents the communication relationship among classes [2, 12]. This class interaction diagram shows the client class `Class_Interaction` and all possible communication interaction initiated by `Class_Interaction`. There are 50 classes that collaborate with `Class_Interaction` through possible message sends. Figure 3 is representative of the complex interactions we observed for the eight programs in this study.

In the class interaction diagram, a node represents a class, and an edge between a node representing class X and a node representing class Y indicates that an object of type X may send a message to an object of type Y. Thus, each edge represents a set of call graph edges between methods of different classes. The construction of the diagram handles polymorphism in a conservative way. This type of diagram is often used in object-oriented analysis and design to illustrate collaborations among classes and to show the structure and relationship between classes in an entire program. Other names for a class interaction diagram are class communication diagram, collaboration diagram, and relationship diagram.

Table 2 gives a short description of the eight programs that we studied. The results of our analysis are given in Figures 4 and 5. In each figure, the first column of charts presents information quantifying the method usage for all classes of an application. In each chart, the height of the bar labeled *i-j* indicates the number of classes in the application with a percentage of potentially invoked methods falling between *i* and *j*, inclusive. A method is counted as potentially invoked if it could be called directly or indirectly by the client, as determined through static analysis. For example, the chart describing the program `Class_Interaction` shows that ten of the server classes use 20-29% of their defined methods, 18 classes use 50-59% of their defined methods, and only three classes use 100% of their defined methods.

The goal of the second column of charts is to reveal the actual space savings of using the server-based approach versus constructing CCFG for each class in the program. In each chart, the height of the bar labeled *i-j* indicates the number of classes in the application with a percentage of server class CFG nodes constructed by the server-based approach falling between *i* and *j*, inclusive. For example, in `Class_Interaction`, eight of the server classes have 20-29% of their CFG nodes constructed as part of the server-based approach. While these percents are affected by the percents of potentially invoked methods, these statistics give more insight into the size of the methods that are potentially invoked.

¹Code Sources:

JGL: Java Generic Library www.objectspace.com/jgl,
CAL: Container and Algorithm Library www.x3m.com/products/cal,
SJL: Simple Java(tm) Library www.pip.dknet.dk/pip1848/sjl/sjl.htm,
BIT: Bytecode Instrumenting Tool www.cs.colorado.edu/~hanlee/BIT,
JTREK: www.digital.com

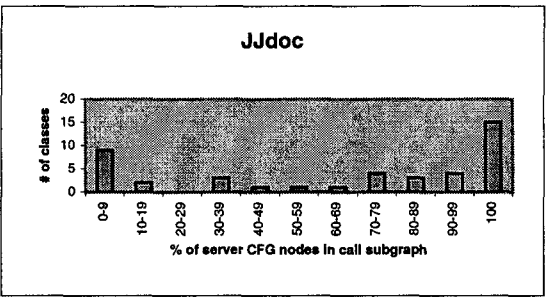
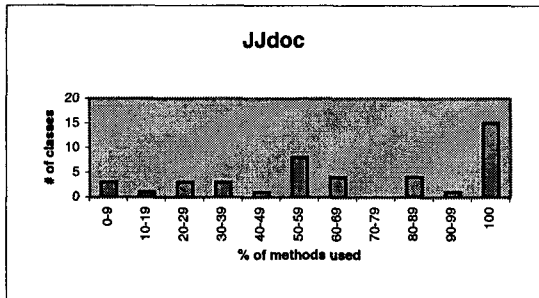
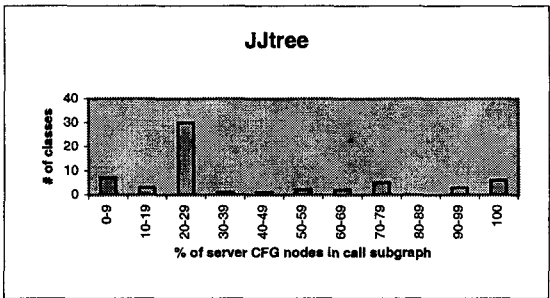
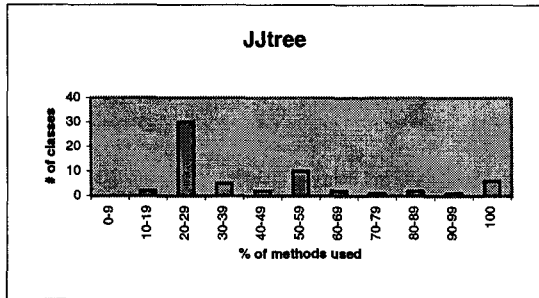
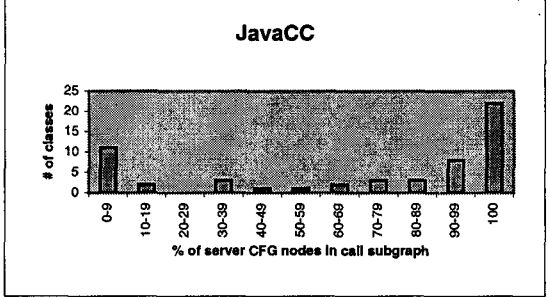
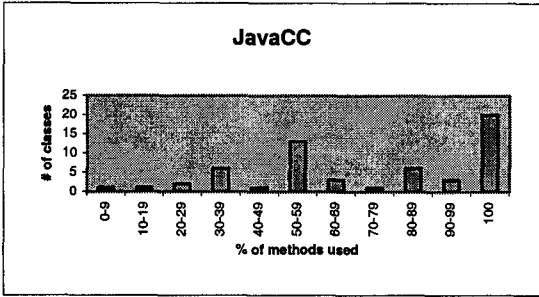
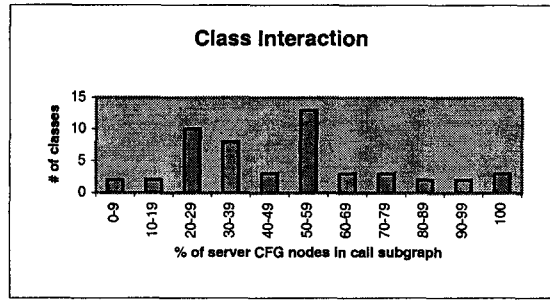
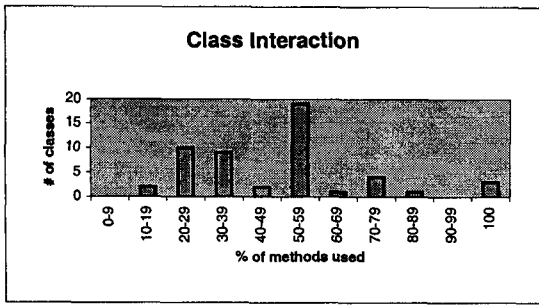


Figure 4: Call subgraph and CCFG sizes in larger applications.

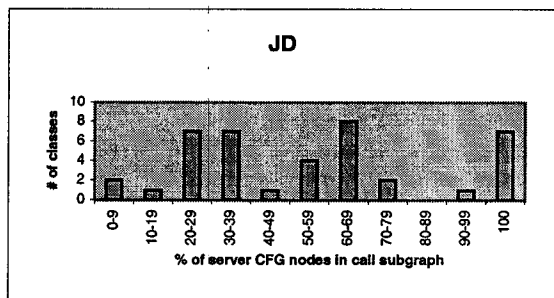
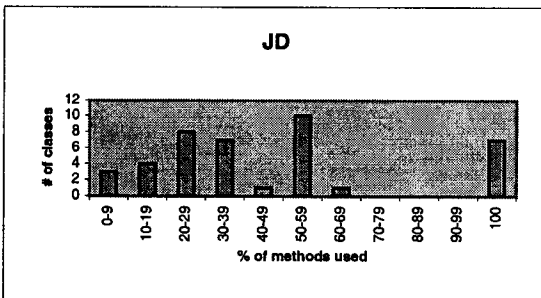
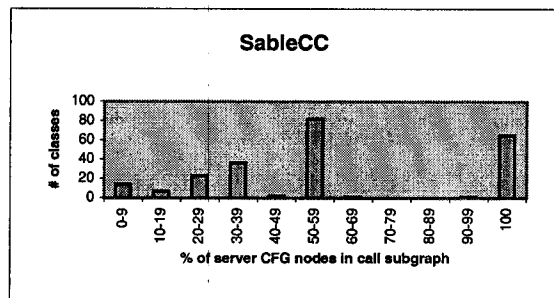
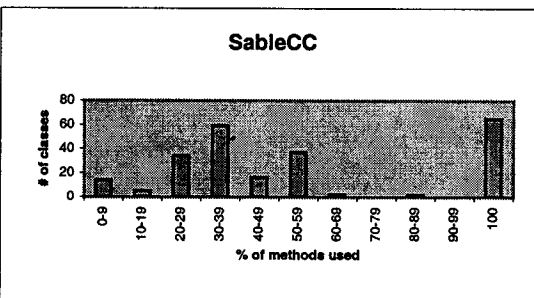
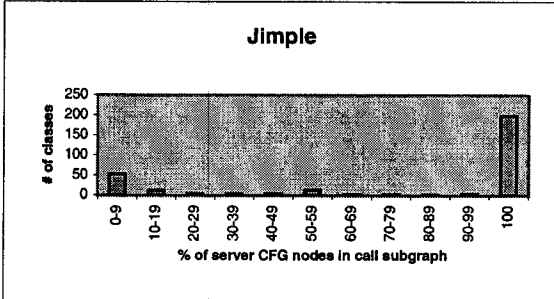
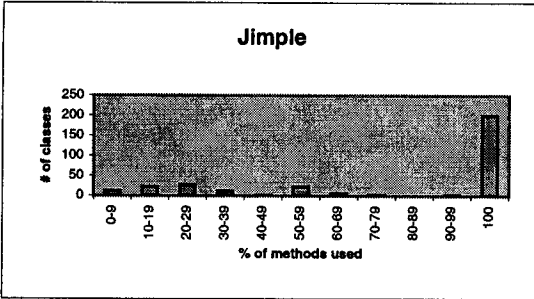
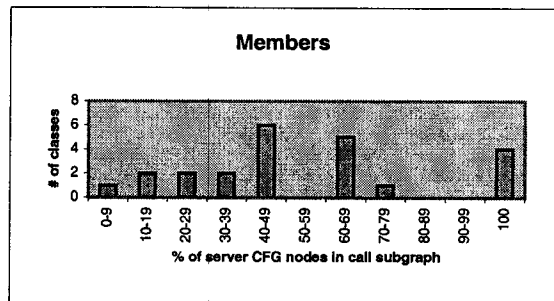
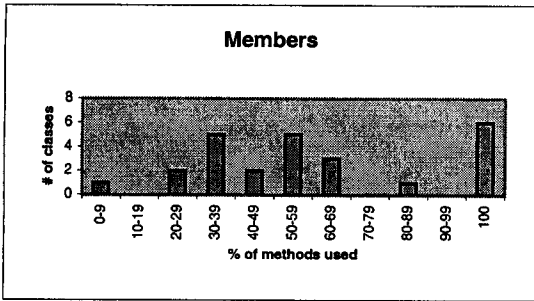


Figure 5: Call subgraph and CCFG sizes in larger applications continued.

From these charts, we can conclude (similar to the conclusions for the smaller benchmarks) that a relatively small number of server classes have all of the methods defined in a class actually appearing as part of the call subgraph. Many of the classes have fewer than 60% of their methods as part of the call subgraph. The second column of charts suggests that the amount of savings due to building only the CFGs corresponding to potentially invoked methods could be significant for all applications and quite large for applications, like `Class_Interaction`, `SableCC`, `Members`, and `JD`. The small size of methods and the branch counts, coupled with the observations of method usage in both small and large applications led us to examine approaches to def-use analysis of a class that perform analysis of its server classes with less emphasis on control flow effects, and consequently do not require as much program representation for the server class when analyzing a client class.

3.3 Client-based Algorithm

For the problem of computing def-use pairs for a class that instantiates and uses an object of another class, we found that we need to refine McGregor et al [13]’s definitions of def and use of an object, as well as the definition of a modifier method.

An *explicit modifier* of class T is a method that contains an explicit definition of an instance variable of class T. An *implicit modifier* of class T is a method that may modify an instance variable of class T indirectly through a call chain leading to an explicit modifier of class T. We use the term modifier to indicate either explicit or implicit modifier method. Thus, the definition of a modifier can be rephrased in terms of the flow insensitive side effect problem, *mayMOD*. The *mayMOD* set for a method is the set of instance variables that may be modified by executing that method from any call site within the call subgraph [14]. A method M is considered to be a *modifier* if $mayMOD(M)$ is nonempty.

We have similar definitions for *explicit reader*, *implicit reader*, and *reader*. A method M is considered to be a *reader*, either implicit or explicit, if $mayREF(M)$ is nonempty, where $mayREF(M)$ is the set of instance variables that may be referenced by executing M from any call site in the call subgraph [14]. Figure 6 shows the call subgraph for the `Array` class of Figure 2 rooted at methods within class `genericStack`. Each node is labeled with its modifier and reader status. Due to space limitations, the code example is very simple, and thus the call subgraph has few edges between nodes within the server class.

We now define the *def* and *use* of an object as follows:

Let c be a call site in a method of class C, and let (M^0, M^1, \dots, M^n) be the set of methods of S which potentially could be called at c .

Def of an object of class S located in class C: c is said to be a def of the object of class S if $mayMOD(M^i)$ is nonempty for any $M^i, 0 \leq i \leq n$.

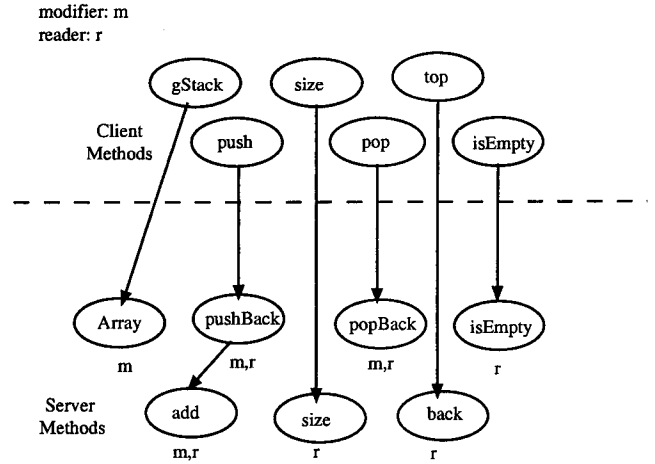


Figure 6: Server call subgraph with modifiers and readers.

Use of an object of class S located in class C: c is said to be a use of the object of class S if $mayREF(M^i)$ is nonempty for any $M^i, 0 \leq i \leq n$.

With these more refined definitions, we first compute *mayMOD* and *mayREF* flow insensitive data flow information over the call subgraph of S. The methods called from C are then found to be either a def, use, or both def and use based on the *mayMOD* and *mayREF* information. The CCFG is built only for C, and intra-method, inter-method, and intra-class def-use object pairs are computed using this def and use information at callsites.

With the client-based approach, def-use analysis of a client class C’s use of objects of another class S computes intra-method, inter-method, and intra-class def-use pairs of object types within C by computing the def-use pairs of call sites in C to the server class. This approach does not construct the CCFG for any part of class S, and thus, saves in program representation space and construction time. In the client-based approach, none of the CFG nodes in the last column of Table 1 are constructed.

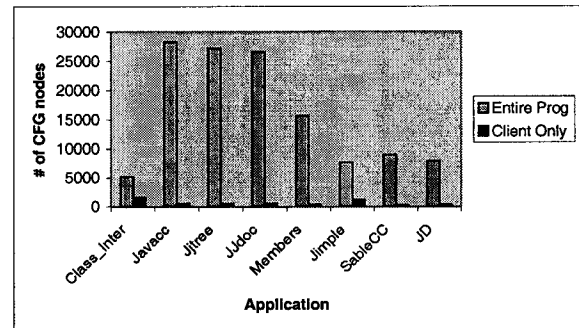


Figure 7: Program representation size comparison.

Figure 7 shows the amount of savings in terms of the size of the program representation by only building the CCFG of the client class, instead of building the CCFG for all the classes in our eight Java programs. The first column above the application name indicates the size of the CCFG program representation for the entire program, while the second column shows the size of the CCFG program representation for only the client. For example, JavaCC's entire representation consists of 28,332 CCFG nodes, whereas the client's CCFG contains only 537 nodes.

The tradeoff between client-based and server-based approaches is that the inter-class def-use information computed for the client class is flow insensitive within its server classes. However, we believe that the nature of object-oriented design characteristics that we have observed regarding the simple control flow within methods, small size of methods, and the small percentage of modifiers, shows that this information could be adequate for some software engineering applications. For example, in data flow testing, def-use coverage of a class's use of object types would consist of covering the appropriate pairs of call sites in the class being tested.

4 Conclusions and Future Directions

We have examined the problem of computing (intra-method, inter-method, and intra-class) def-use pairs for a class's manipulation of objects of other classes. Code reuse through developing server classes that are used by many client classes causes this situation to be a common occurrence in object-oriented programs, but one which has not been adequately addressed by existing data flow analysis techniques. This paper presents evidence of the large number of and complexity of interactions of server classes created through a single client class. We propose an approach that avoids creating large program representations, by using flow insensitive analysis of the server classes. While our study of the characteristics of client-server class relationships leads us to believe that this approach will give useful information for software engineering applications, we are currently in the midst of an experimental investigation, and also examining some variations of our approach.

References

- [1] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, 1996.
- [2] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.
- [3] Ramkrishna Chatterjee and Barbara Ryder. Data-flow-based Testing of Object-Oriented Libraries. Technical Report 382, Rutgers University, March 1999.
- [4] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant Context Inference. In *Proceedings of Principles of Programming Languages*, 1999.
- [5] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, 1996.
- [6] R. Gupta, M.J. Harrold, and M.L. Soffa. An Approach to Regression Testing using Slicing. In *Proceedings of Software Maintenance*, pages 299–308, 1992.
- [7] Mary Jean Harrold and Gregg Rothermel. Performing Data Flow Testing on Classes. In *Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1994.
- [8] M.J. Harrold and M.L. Soffa. Interprocedural Data Flow Testing. In *Proceedings of the ACM Symposium on Testing, Analysis, and Verification*, pages 158–167, 1989.
- [9] S. Horwitz, T. Reps, and D. Binkley. Intergrating Non-interfering Version of Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, 1989.
- [10] Loren Larsen and Mary Jean Harrold. Slicing Object-Oriented Software. In *Proceedings of the International Conference on Software Engineering*, 1996.
- [11] Donglin Liang and Mary Jean Harrold. Slicing Objects Using System Dependence Graphs. In *Proceedings of the International Conference on Software Maintenance*, pages 358–367, 1998.
- [12] James Martin. *Principles of Object-Oriented Analysis and Design*. Prentice Hall, 1993.
- [13] John D. McGregor, Brian A. Malloy, and Rebecca L. Siegmund. A Comprehensible Program Representation of Object-Oriented Software. *Annals of Software Engineering*, 2, 1996.
- [14] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [15] H. Pande, W. Landi, and B.G. Ryder. Interprocedural Def-Use Associations in C Programs. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.
- [16] T. Reps, T. Teitelbaum, and A. Demers. Incremental Context-Dependent Analysis for Language Based Editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.
- [17] Frank Tip, Jong-Deok Choi, John Field, and G. Ramalingam. Slicing Class Hierarchies in C++. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, 1996.
- [18] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.