Performing Data Flow Testing on Classes*

Mary Jean Harrold and Gregg Rothermel
Department of Computer Science
Clemson University
Clemson, SC 29634-1906
{harrold, grother}@cs.clemson.edu

Abstract

The basic unit of testing in an object-oriented program is a class. Although there has been much recent research on testing of classes, most of this work has focused on black-box approaches. However, since black-box testing techniques may not provide sufficient code coverage, they should be augmented with code-based or white-box techniques. Dataflow testing is a code-based testing technique that uses the dataflow relations in a program to guide the selection of tests. Existing dataflow testing techniques can be applied both to individual methods in a class and to methods in a class that interact through messages, but these techniques do not consider the dataflow interactions that arise when users of a class invoke sequences of methods in an arbitrary order. We present a new approach to class testing that supports dataflow testing for dataflow interactions in a class. For individual methods in a class, and methods that send messages to other methods in the class, our technique is similar to existing dataflow testing techniques. For methods that are accessible outside the class, and can be called in any order by users of the class, we compute dataflow information, and use it to test possible interactions between these methods. The main benefit of our approach is that it facilitates dataflow testing for an entire class. By supporting dataflow testing of classes, we provide opportunities to find errors in classes that may not be uncovered by black-box testing. Our technique is also useful for determining which sequences of methods should be executed to test a class, even in the absence of a specification. Finally, as with other code-based testing techniques, a large portion of our technique can be automated.

*This work was partially supported by NSF under Grants CCR-9109531 and CCR-9357811 to Clemson University.

1 Introduction

One of the most important benefits of object-oriented programming is the ability to reuse classes. A *class* is an instantiable, information-hiding module that defines the data (*instance variables*) and operations (*methods*) that an object of that class will contain[1]. A class is often considered to be the basic *unit* of testing in an object-oriented program. Although there has been much research recently on testing of classes, most of this work has focused on black-box techniques [4, 11, 12, 21] that do not use the class's code to select tests. However, since black-box techniques may not provide sufficient coverage of the code, these techniques should be augmented with code-based or white-box approaches.

One type of code-based testing is *dataflow testing* [2, 6, 9, 15, 16, 17, 20], which uses the dataflow relationships in a program to guide the selection of tests. Several dataflow testing tools have been developed [5, 7, 13, 17], and some of these tools have been used to investigate the effectiveness of dataflow testing in uncovering program errors [3, 14, 22]. However, no technique has been presented that applies dataflow testing to object-oriented programs. Existing dataflow testing techniques can be applied to both individual methods and methods in a class that interact through messages (procedure calls between methods), but these techniques do not consider the dataflow interactions that arise when users of a class invoke sequences of methods in an arbitrary order.

To address this problem, we present a new approach that supports dataflow testing for all types of dataflow interactions in a class. For individual methods in a class, and methods that interact with other methods in the class through procedure calls, our technique is similar to existing dataflow testing techniques [6, 9, 15, 16, 17, 20]. For methods that are accessible outside the class, and can be called in any order by users of the class, we compute dataflow information, and use it to test the possible interactions between these methods. To compute this dataflow information, we develop a new graph representation for a class, the *class control flow graph*, that "connects" all methods in the class. We then adapt an existing dataflow analysis algorithm [18]

[1]When we test a class, we actually test the objects that are instantiations of that class. However for our discussion we use class and object synonymously.

to the class control flow graph to compute the dataflow information required for dataflow testing.

The main benefit of our approach is that we provide a way to use dataflow testing to test an entire class. By supporting dataflow testing of classes, we provide opportunities to find errors in classes that may not be uncovered by black-box testing. Our technique is also useful for determining which sequences of methods should be executed to test a class, even in the absence of a specification. When used in conjunction with black-box techniques, our technique also helps reduce testing of unnecessary sequences of methods, since it provides information about sequences in which methods do not interact. Finally, as with other code-based testing techniques, a large portion of our technique can be automated.

In the next section, we briefly overview both dataflow analysis and dataflow testing. In Section 3, we describe class testing and the application of dataflow testing to classes. Section 4 outlines our requirements for dataflow testing of classes. In Section 5, we present our technique for computing the dataflow information required for dataflow testing of classes. Section 6 discusses other issues relevant to dataflow testing of classes, Section 7 compares our work with related work, and Section 8 presents our conclusions.

2 Dataflow Analysis and Testing

In dataflow testing [6, 15, 16, 20], an assignment to a variable in a program is tested by executing subpaths from the assignment (*definition*) to points where the variable is used (*use*). Uses in the program are either *computation* uses (c-uses) or *predicate* uses (p-uses) [20]. A c-use occurs whenever a value is used in a computation or output statement; a p-use occurs whenever a value is used in a predicate statement. A *definition-use pair* (def-use pair) is an ordered pair (d, u), where d is a statement containing a definition of a variable v and u is a statement containing a use of v, or some memory location bound to v, that can be reached by d over some path in the program.

Test data adequacy criteria are used to select particular def-use pairs or subpaths that are identified as the test requirements for a program; tests are generated that, when used in a program's execution, satisfy these requirements. A test satisfies a def-use pair if executing the program with the test causes traversal of a subpath from the definition to the use without any intervening redefinition of that memory location. For c-uses, traversal must be from the statement containing the definition to the statement containing the use, while for p-uses, tests must traverse subpaths from the statement containing the definition to both successors of the statement containing the use. Many different dataflow testing criteria have been defined and compared [2, 6, 15, 16, 20]. One criterion, the 'all-uses' criterion, requires that each feasible def-use pair in the program be tested[2]. The 'all-uses' criterion has been shown to be practical since typically, relatively few tests are required for its satisfaction [23].

When dataflow testing is used for unit testing of individual procedures, def-use pairs are computed using traditional iterative or interval-based dataflow analysis methods [1]. These analysis methods use a *control flow graph* to represent the program. Nodes in a control flow graph represent program statements and edges represent the flow of control between statements; each graph is augmented with a single *entry* node and a single *exit* node. To apply dataflow testing to interacting procedures [9], and to test C programs [13, 17], more precise dataflow analysis is required. Interprocedural dataflow analysis [10] computes def-use pairs whose definition is in one procedure and whose use is in a called or calling procedure. This technique computes def-use pairs for global variables and reference parameters using a form of the call graph to propagate dataflow information throughout the program. Dataflow analysis for C programs [18] also considers the effects of pointer variables and aliasing in computing the def-use pairs for testing; for discussion, we call this technique, developed by Pande, Landi and Ryder, the PLR algorithm.

The PLR algorithm constructs an *interprocedural control flow graph* for a program, which combines control flow graphs of individual procedures. Each call site is replaced by a *call* and a *return* node. The control flow graphs are connected by adding edges from call nodes to entry nodes and from exit nodes to return nodes, to represent procedure calls in the program. A special entry node ρ is used to represent the entry to the "main" procedure of the program. The PLR algorithm first computes conditional alias and definition information for each procedure. Then, using a dataflow framework, it propagates the local information to obtain interprocedural reaching definitions from which interprocedural def-use pairs are calculated.

3 Classes and Class Testing

A class defines the data relevant to an object of that class, and a set of operations that may be performed on that data; class data are referred to as *instance variables*, and class operations are called *methods*. Without loss of generality, we use a model for a class that contains only two levels of access to the class's instance variables and methods: *public* and *private*. Public instance variables and methods can be accessed by users of the class, while private instance variables and methods are accessible only within the class. A class may contain a *constructor* method that is executed whenever an object of that class is instantiated, and a *destructor* method that is executed when the object is destroyed. Figure 1 presents a partial C++ listing for a `SymbolTable` class, which serves as an example throughout this paper. The `SymbolTable` class contains public methods `SymbolTable` (the constructor), `~SymbolTable` (the destructor), `AddtoTable`, and `GetfromTable`, and private methods `Lookup`, `Hash`, `GetSymbol`, `GetInfo`, `AddSymbol`, and `AddInfo`.

[2] A def-use pair is *feasible* if there is some program input that will cause it to be executed, and *infeasible* otherwise.

```
1  // symboltable.h: definition of SymbolTable class
2  #include "symbol.h"
3
4  class SymbolTable {
5  private:
6     TableEntry *table;
7     int numentries, tablemax;
8     int *Lookup( char *);
9  public:
10    SymbolTable(int n) {
11        tablemax = n;
12        numentries = 0;
13        table = new TableEntry[tablemax]; };
14    ~SymbolTable() { delete table; };
15    int AddtoTable(char *symbol, char *syminfo);
16    int GetfromTable(char *symbol, char *syminfo);
17 };
18
19 // symboltable.c: implementation of SymbolTable class
20 #include "symboltable.h"
21
22 int SymbolTable::Lookup(char *key, int index) {
23    int saveindex;
24    int Hash(char *);
25    saveindex = index = Hash(key);
26    while ( strcmp(GetSymbol(index),key) != 0) {
27       index++;
28       if (index == tablemax) /* wrap around */
29          index = 0;
30       if (GetSymbol(index)==0 || index==saveindex)
31          return NOTFOUND;
32    }
33    return FOUND;
34 }
35
36 int SymbolTable::AddtoTable(char *symbol,
                              char *syminfo) {
37    int index;
38    if (numentries < tablemax) {
39       if (Lookup(symbol,index) == FOUND)
40          return NOTOK;
41       AddSymbol(symbol,index);
42       AddInfo(syminfo,index);
43       numentries++;
44       return OK;
45    }
46    return NOTOK;
47 }
48
49 int SymbolTable::GetfromTable(char *symbol,
                                 char **syminfo) {
50    int index;
51    if (Lookup(symbol,index) == NOTFOUND)
52       return NOTOK;
53    *syminfo = GetInfo(index);
54    return OK;
55 }
56
69 void SymbolTable::AddInfo(syminfo,index) {
70     . . .
73     strcpy(table[index].syminfo,syminfo);
75 }
76
77 char *SymbolTable::GetInfo(index) {
78     . . .
82     return table[index].syminfo;
83 }
```

Figure 1: Partial listing for the `SymbolTable` class.

We represent the call structure of a class using a *class call graph*. A class call graph is a directed graph in which nodes represent methods, and edges represent procedure calls between methods. Figure 2 displays the class call graph for class `SymbolTable`. Because `GetfromTable` calls `Lookup` and `GetInfo`, the graph contains edges from the `GetfromTable` node to the `Lookup` and `GetInfo` nodes. Figure 2 also depicts edges, shown as dashed lines, ending at each of the class's public methods. The dashed edges represent messages sent to these public from outside the class.

We test classes at three "levels", which we define as follows:

Intra-method testing tests methods individually. This level of testing is equivalent to unit testing of individual procedures in procedural-language programs.

Inter-method testing tests a public method together with other methods in its class that it calls directly or indirectly. This level of testing is equivalent to integration testing of procedures in procedural-language programs.

Intra-class testing tests the interactions of public methods when they are called in various sequences. Since users of a class may invoke sequences of methods in indeterminate order, we use intra-class testing to increase our confidence that sequences of calls interact properly. However, since the set of possible public method call sequences is infinite, we can only test a subset of this set.

To illustrate each of these levels of testing, consider the `SymbolTable` class, shown in Figure 1. We perform intra-method testing on the `SymbolTable` class by testing each of the ten methods in the class separately. We perform inter-method testing on the `AddtoTable` method by integrating the `AddtoTable`, `AddSymbol`, `Lookup`, `AddInfo`, `GetSymbol`, and `Hash` methods, and testing various calls to `AddtoTable`. Similarly, we perform inter-method testing on the `GetfromTable` method by integrating the `GetfromTable`, `Lookup`, `GetInfo`, `GetSymbol`, and `Hash` methods, and testing various calls to `GetfromTable`. Since the constructor and destructor methods in the `SymbolTable` class (`SymbolTable` and ~`SymbolTable`) do not call other methods, intra- and inter-method testing of these routines are equivalent. For intra-class testing, we may select test sequences such as <`SymbolTable`, `AddtoTable`, `GetfromTable`> and <`SymbolTable`, `AddtoTable`, `AddtoTable`>. In all cases, we may require stubs and/or driver procedures to perform the testing; we may also require a technique to inspect the state of the object after the method sequence invocation.

Previous research on class testing has focused on techniques for selecting sequences of methods for intra-class testing[3, 11, 12, 19, 21]. However, existing techniques suffer from two drawbacks. First, most existing techniques select tests on the basis of specifications, or from state graphs constructed solely from specifications. Since most software systems are not formally specified,

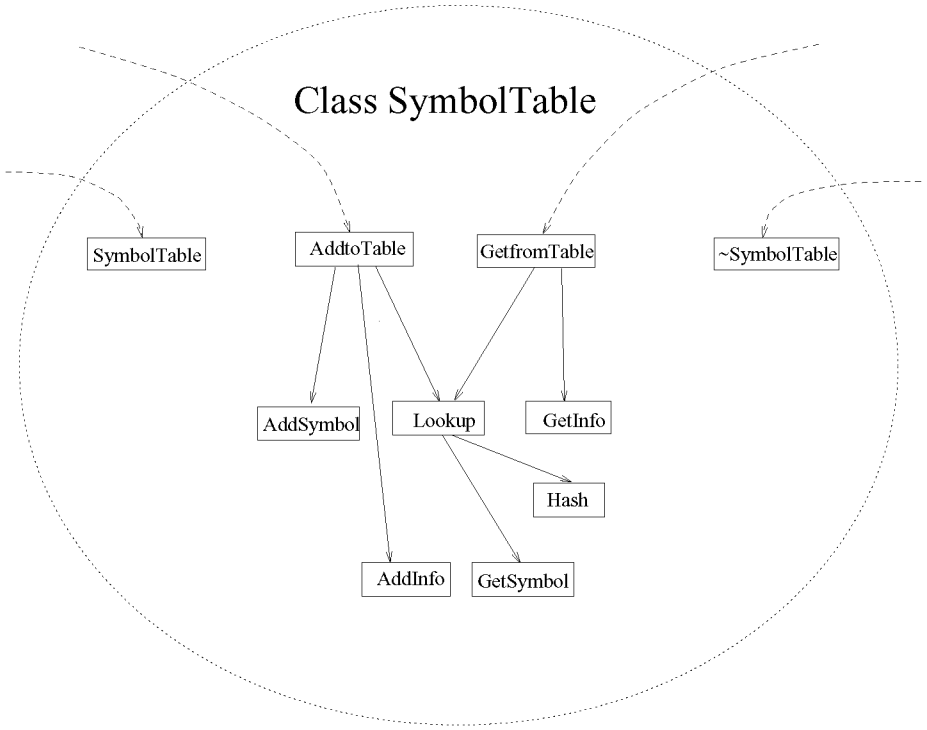

Figure 2: Class call graph for the `SymbolTable` class.

and many are not even informally specified, these methods are frequently not applicable. Furthermore, even when a program is specified, specification-based testing may not detect errors caused by implementation details not addressed in the specifications. For example, if the specification for `SymbolTable` does not require that `table` be implemented as an array and maintained by linear hashing, we cannot know from the specifications that we should test whether the table search method, `Lookup`, successfully wraps around (line 29).

The second drawback of existing intra-class testing techniques, including code-based techniques, is that they simply specify sequences of methods to execute, without requiring coverage of particular code components. However, we would not have confidence in our class testing if the class contains code that has not been exercised sufficiently.

In Section 7 we discuss related research in greater detail.

4 Dataflow Testing of Classes

To supplement the existing intra-class testing techniques, we require a code-based testing technique that identifies components of a class that should be tested. One such technique is dataflow testing, which considers all class variables, and program-point-specific def-use pairs. There are three types of def-use pairs in classes that should be tested. These types correspond to the three levels of testing defined above. We define the three types formally first, and then describe them informally and with examples.

In the following definitions, let C be the class being tested. Let d represent a statement containing a definition, and u represent a statement containing a use.

Intra-method def-use pairs. Let M be a method in C. If d and u are both in M, and there exists a program P that calls M such that in P, *(d,u)* is a def-use pair exercised during a single invocation of M, then *(d,u)* is an intra-method def-use pair.

Inter-method def-use pairs. Let M_0 be a public method in C, and let $\{M_1, M_2, ..., M_n\}$ be the set of methods in C called, directly or indirectly, when M_0 is invoked. Suppose that d is in M_i and u is in M_j, where both M_i and M_j are in $\{M_0, M_1, M_2, ..., M_n\}$. If there exists a program P that calls M_0 such that in P, *(d,u)* is a def-use pair exercised during a single invocation by P of M_0, and such that either $M_i \neq M_j$, or M_i and M_j are separate invocations of the same method, then *(d,u)* is an inter-method def-use pair.

Intra-class def-use pairs. Let M_0 be a public method in C, and let $\{M_1, M_2, ..., M_n\}$ be the set of methods in C called, directly or indirectly,

when M_0 is invoked. Let N_0 be a public method in C (possibly the same method as M_0), and let $\{N_1, N_2, ..., N_n\}$ be the set of methods in C called, directly or indirectly, when N_0 is invoked. Suppose d is in some method in $\{M_0, M_1, M_2, ..., M_n\}$, and u is in some method in $\{N_0, N_1, N_2, ..., N_n\}$. If there exists a program P that calls M_0 and N_0, such that in P, *(d,u)* is a def-use pair, and such that after d is executed and before u is executed, the call to M_0 terminates, then *(d,u)* is an inter-method def-use pair.

Informally, intra-method def-use pairs occur within single methods, and test def-use interactions that are limited to those methods. For example, in the `SymbolTable` class, the `Lookup` method contains intra-method def-use pair (27,28) with respect to variable `index`, because the definition of `index` in node 27 reaches the use of `index` in node 28.

Inter-method def-use pairs occur when methods within the calling context of a single public method interact, such that a definition in one method reaches across method boundaries to a use in some method called, directly or indirectly, by the public method. For example, in the `SymbolTable` class, public method `AddtoTable` invokes the `Lookup` method, and receives an index value back, which it uses in the call to `AddSymbol`. Def-use pair (29,41) is an inter-method pair, because the definition of `index` at line 29 in `Lookup` reaches the use of `index` at line 41 in `AddtoTable`.

Finally, intra-class def-use pairs occur when sequences of public methods are invoked. For example, consider the method sequence <`AddtoTable`, `AddtoTable`>. In the first call to `AddtoTable`, if a symbol is added to `table`, line 43 sets `numentries`. In the second call to `AddtoTable`, line 38 fetches the value of `numentries`. Thus (43,38) is an intra-class def-use pair. As a second example, consider the sequence <`AddtoTable`, `GetfromTable`>. `AddtoTable` may add symbol information to `table`, by calling the `AddInfo` routine; `GetfromTable` then accesses `table` by calling `GetInfo`. The definition of the table entry at line 73 in `AddInfo` and the use of the table at line 82 in `GetInfo` form an intra-class def-use pair.

All three types of def-use pairs are useful for testing classes. For example, if we use the 'all-uses' dataflow coverage criteria, then intra-method def-use pair (27,28) tests whether `Lookup` successfully "resets" to the start of the table when it reaches the end of the table array[3]. Inter-method def-use pair (29,41) tests whether we can add a symbol at the 0th location of `table`. Intra-class def-use pair (43,38) tests whether `AddtoTable` acts correctly when `table` is full. Finally, intra-class def-use pair (73,82) tests whether information added to the symbol table can be fetched.

Intra-class pairs have the further advantage of guiding testers in the selection of sequences of methods that should be run, and sequences of methods that need not be run. For example, to exercise the intra-class pair (73,82) we must test method sequence <`AddtoTable`, `GetfromTable`>. However, there are no intra-class pairs originating within the `GetfromTable` method (or methods it invokes) and terminating within the `AddtoTable` method (or methods it invokes). This suggests that `GetfromTable` cannot affect `AddtoTable`, and that we do not need to test method sequence <`GetfromTable`,`AddtoTable`>.

To further illustrate the advantages of intra-class dataflow testing, suppose the conditional in line 38 of the `SymbolTable` program is changed (erroneously) to the following statement.

```
if (numentries <= tablemax)
```

Suppose we have `tablemax`+1 distinct symbols, and we insert all but one into `table` by making `tablemax` calls to `AddtoTable`. Now suppose we call `AddtoTable` with the final symbol, "s". On this call, `numentries` equals `tablemax`, so `AddtoTable` enters its outermost `then` clause and calls `Lookup` at line 39. `Lookup` does not find "s" in `table`, but finds no empty table locations either. Thus, after examining every table entry, `Lookup` exits because of the test at line 30 and returns `NOTFOUND`. `AddtoTable` then inserts "s" at the location pointed to by `index`, overwriting the table entry currently in that location.

To detect this fault, we must make `tablemax`+1 calls to `AddtoTable`. Recall that class `SymbolTable` contains intra-class def-use pair (43,38). To achieve 'all-uses' adequacy for this pair, we must find a sequence (or sequences) of method invocations that execute(s) line 43 and then line 38, such that both branches from 38 are taken. By calling `AddtoTable` twice, we exercise (43,38) and take the `true` branch from 38. However, to force the `false` branch to be taken, we must first fill the table by making `tablemax` calls to `AddtoTable`, and then make an additional call to `AddtoTable`. In other words, if we require 'all-uses' testing of pair (43,38), we necessarily execute the sequence of methods that exposes the fault.

5 Computing Dataflow Information for Classes

To support dataflow testing of classes, we must compute all types of def-use pairs for classes; we require intra-method, inter-method, and intra-class def-use pairs. The PLR algorithm [18], described in Section 2, can be used to compute intra/inter-method def-use pairs, but cannot be used directly to compute intra-class def-use pairs, because it requires a complete program from which to construct an interprocedural control flow graph. To compute intra-class def-use pairs, we must consider the interactions that occur when sequences of public methods are invoked. We require a graph that lets us consider all such interactions, and lets us apply algorithms such as PLR.

[3]Recall from Section 2 that the 'all-uses' criteria requires us to test each feasible def-use pair in a program. Testing a def-use pair when the use is a p-use requires that we test both branches out of the p-use.

```
algorithm   ConstructCCFG(C):G
input       C: a class.
output      G: the CCFG for C
declare     frame : set of frame nodes and edges
begin ConstructCCFG
    /* Step 1: Construct the class call graph for the class */
       G = Construct the class call graph for C

    /* Step 2: Add the frame to the class call graph */
       G = G ∪ frame

    /* Step 3: Replace each call graph node with the corresponding control flow graph */
       foreach method M in C do
            Replace M's class call graph node in G with M's control flow graph
            Update edges appropriately

    /* Step 4: Replace call sites with call and return nodes */
       foreach call node S in G, representing a call to method M in C do
            Replace S with a call and a return node
            Update edges appropriately

    /* Step 5: Connect the individual control flow graphs */
       foreach method M in G do
            Add an edge from frame call node to the entry node of M's control flow graph in G
            Add an edge from the exit node of M's control flow graph in G to the frame return node

    /* Step 6: Return the completed class control flow graph G */
       return G

end ConstructCCFG
```

Figure 3: Algorithm to construct a class control flow graph (CCFG).

To compute all three types of def-use pairs for a class, we construct a *class control flow graph* (CCFG). Our algorithm, `ConstructCCFG`, for constructing a CCFG, is given in Figure 3. `ConstructCCFG` inputs a class `C` and outputs `G`, the CCFG for `C`. In Step 1 of `ConstructCCFG`, we construct the class call graph for `C` and initialize `G` to this graph. Step 2 encloses the class call graph in a *frame*, which facilitates the dataflow analysis for the class. A frame represents a driver for the class that lets us simulate arbitrary sequences of calls to public methods. A frame contains five nodes: *frame entry* and *frame exit*, which represent entry to and exit from the frame, respectively; *frame loop*, which facilitates sequencing of methods; and *frame call* and *frame return* nodes, which represent the call to and return from any public method, respectively. A frame also contains four edges: (frame entry, frame loop), (frame loop, frame call), (frame loop, frame exit), and (frame return, frame loop).

Figure 4 shows the class call graph for the `SymbolTable` class along with its enclosing frame. In the figure, frame nodes are shaded and frame edges are shown as dashed lines. Note that at this point in the construction of the CCFG, the frame and the class call graph are not connected.

In Step 3 of `ConstructCCFG`, we replace each class call graph method node `M` in the partially completed CCFG `G` with the control flow graph for `M`; we also update the class call graph edges in `G` so that there are edges from call sites to entry nodes and from exit nodes back to call sites. Then, in Step 4, we replace each call site `S` in `G` with a call node and a return node; we update the class call graph edges in `G` so that there are now edges from call nodes to entry nodes and from exit nodes to return nodes. In Step 5, we connect the frame with the rest of the graph that it encloses. To do this, we add edges from the frame call node to the entry node of each public method, and add edges from the exit of each public method to the frame return node. Finally, in Step 6, we return the completed class control flow graph `G`.

In Figure 5, we show part of the CCFG for the `SymbolTable` class; portions of the graph shown in dotted boxes are not expanded in the figure. Consider the parts of the CCFG that represents methods `AddtoTable` and `Lookup`. The call site to `Lookup` in `AddtoTable` has been replaced by a call node, marked **C**, and a return node, marked **R**.. There are edges from **C** in `AddtoTable` to the entry node in `Lookup` and from the exit node in `Lookup` to **R** in `AddtoTable`. In the complete CCFG,

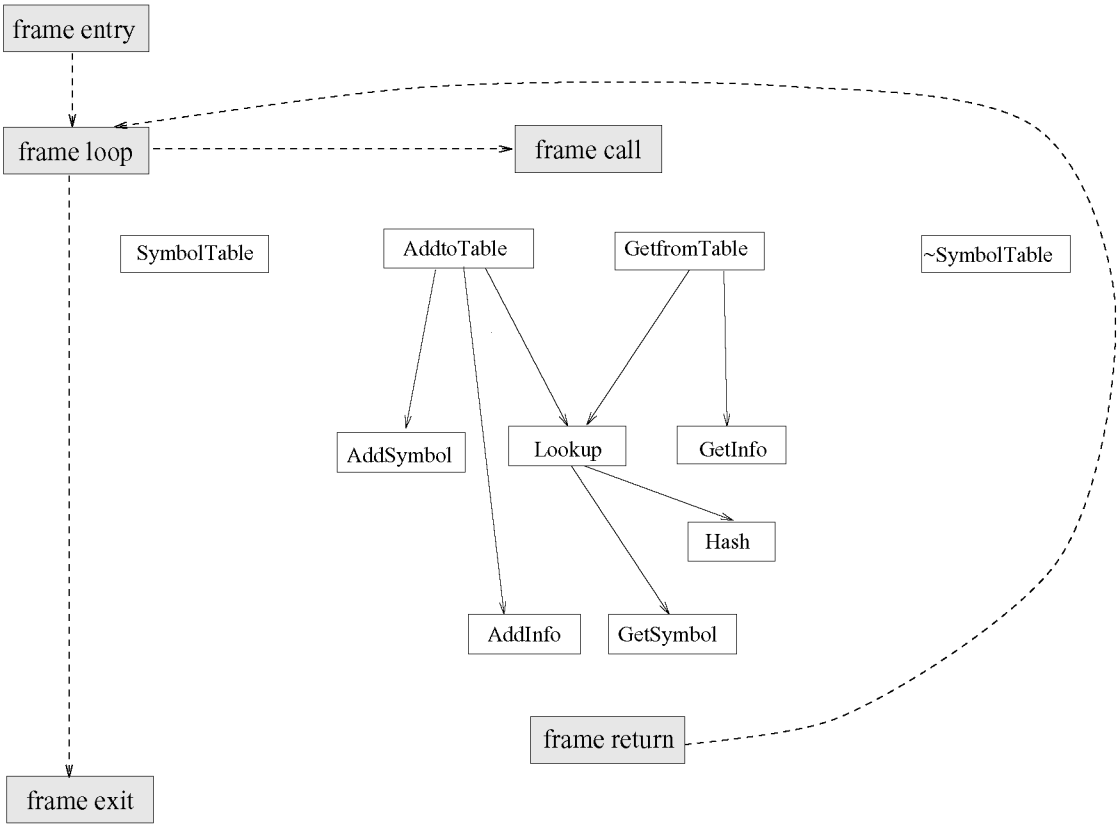

Figure 4: Frame enclosing the class call graph for class `SymbolTable`.

other call sites, such as those at statements 41 and 42 in `AddtoTable`, are also replaced by call and return nodes.

A frame is an abstraction of a main program P in which calls to public methods are selected randomly by a switch statement S, where S is enclosed in a loop L. For programs without the possibility of aliasing[4], existing interprocedural dataflow analysis techniques[9, 18] can be used to get all def-use pairs in the class. When programs contain aliases, analysis methods that do not account for aliasing effects are imprecise in their identification of def-use pairs. For such programs, we can use the PLR algorithm on the CCFG to identify def-use pairs more precisely. Since P simulates a single entry program in which all sequences of message calls are possible, and since the PLR algorithm functions on single entry programs, it follows that the PLR algorithm can be applied to the CCFG. However, applied to the CCFG, the PLR algorithm may miss some def-use pairs that occur in the context of applications programs, when those programs introduce specific aliases. We discuss this problem in Section 6.

To apply the PLR algorithm to the CCFG for C, we do the following. First we compute *conditional alias* and *conditional reaching definitions* information for C in the CCFG. We then propagate the dataflow information throughout the program using the CCFG and the propagation rules specified in [18], with the following adjustments:

- process the frame call node like a call node
- process the frame return node like a return node
- process the frame loop node as a statement node with no definitions or uses
- process the frame entry and exit nodes like program entry and exit nodes

This analysis yields a set of def-use pairs consisting of intra-method def-use pairs, inter-method def-use pairs, and intra-class def-use pairs. Examples of these def-use pairs are given in Section 3.

Similar approaches let us apply other dataflow analysis algorithms to the CCFG. In general, the precision of the def-use information we compute depends on the precision of the dataflow analysis algorithm we use.

[4] An *alias* occurs when two names for the same memory location are visible at a point in the program.

6 Additional Considerations

There are several additional considerations related to dataflow testing for classes. In this section, we briefly discuss some of them.

First, when we perform dataflow analysis on the CCFG for a class, our analysis may miss some intra-method, inter-method, or intra-class def-use pairs that occur in the context of applications programs, when those programs introduce specific aliases. For example, consider a method `M` in class `C`. Suppose that `M` defines public instance variable `*a` and uses another public instance variable `*b`, and suppose there is a path in `M` from `*a` to `*b` on which neither `*a` nor `*b` is defined, as shown below:

```
      method M
        . . .
n1:   *a :=
        . . .  <== no definition of *a or *b
n2:      := *b
```

Assume that in `C`, there is no point where `*a` and `*b` are aliased. In this case, our analysis using the CCFG will not associate the definition of `*a` at `n1` with the use of `*b` at `n2`. However, suppose a program `P` that uses `C` causes `*a` and `*b` to be aliased and then invokes `M`; on this invocation of `M`, (`n1`,`n2`) is a def-use pair. We are currently exploring ways to save alias and def-use information gathered during our analysis of the CCFG, so that we can easily recognize such def-use pairs when we test a class in the context of other programs and classes.

A similar consideration involves the use of dataflow testing for integration of classes. When class C_1 sends messages to class C_2, we may wish to test dataflow interactions between C_1 and C_2. To do this we define a fourth level of dataflow testing, *inter-class* testing. Inter-class testing considers def-use pairs *(d,u)*, such that *d* is in one class and *u* is in another class, to guide the selection of test cases. For inter-class testing, we can compute def-use pairs using interacting CCFG's. Experimentation is required to determine how well this technique will scale to large programs or many interacting classes.

A third issue involves *derived* classes that are obtained using inheritance. A derived class is formed from a base class, where some methods and data are modified, added or deleted. Experiments suggest that tests and testing information originally used to test a base class can be reused to test a derived class [8]. We can also use incremental dataflow analysis algorithms to update def-use pair information in derived classes, which reduces the cost of analysis.

Finally, we are considering ways to handle object-oriented features such as polymorphism and dynamic binding in our dataflow testing approach. Currently, if we encounter a call to a method that is bound at runtime, we can either (1) test the calling method with all possible called methods, or (2) select some representative from the inheritance hierarchy, and use it for the testing. The first approach is precise but may be impractical; the second approach lets us test the calling method with some, but not every, called method.

7 Related Research

Previous research on class testing addresses intra-class testing, and focuses on selection of method sequences to be tested. Most existing techniques for method sequence selection are based on specifications, or on state diagrams constructed from specifications[3, 11, 12, 21]. As discussed in Section 3, these techniques have drawbacks; code-based test selection techniques are also necessary.

Little attention has been paid to code-based test selection criteria for object-oriented software. Parrish, Borie, and Cordes [19] present a "flow-graph-based" test selection technique, that selects method sequences with or without specifications. Given class C, their technique constructs a graph G for C such that for each public method M in C, G contains node N. Given nodes N_i and N_j in G, there is an edge from N_i to N_j if a user can invoke M_i followed by M_j. If M_i sets a condition c such that when c is "true" ("false"), M_j may be invoked, then edge (N_i,N_j) is called a *control edge* and given label "true" ("false"). We can design tests that cover all nodes and/or edges in G.

This technique incorporates a limited form of dataflow information, involving "types" rather than variables. A method M (node N) contains a definition (use) of type T if M contains a formal parameter of type T. A def-use edge is defined as a triple involving a type T, a node in which T is defined, and a node in which T is used. If we insert def-use edges into G, we may apply dataflow-like coverage criteria to G.

There are two drawbacks to this technique. First, the technique considers only method parameters, ignoring shared variables (variables global within the class). Second, the technique is not fine-grained enough: if v is a variable used in class C, the technique does not ensure that we will test from each (or even *any*) definition of v to each (or even *any*) use of v in class C. These drawbacks cause the technique to miss opportunities for selection of significant tests. For example, given the error that results when line 38 of `SymbolTable` is changed, discussed in Section 4, this technique requires us to test method sequence <`AddtoTable`, `AddtoTable`>, but does not necessarily require us to make `tablemax`+1 calls to `AddtoTable`, and thus may not lead to discovery of the error[5].

8 Conclusions

We have presented a technique for applying dataflow testing to classes. We define three levels of dataflow class testing: (1) intra-method testing, which

[5] This technique also allows graphs to be constructed from specifications. If we do this in the case of this example, we may detect this error.

tests individual class methods, (2) inter-method testing, which tests methods in a class that interact through procedure calls, and (3) intra-class testing, which tests sequences of calls to methods. To identify def-use pairs for these three levels of testing, we represent a class as a single-entry, single-exit program, and construct a class control flow graph for this program. We then perform interprocedural dataflow analysis on this graph. In this paper we adapt an existing interprocedural dataflow analysis algorithm [18] to perform analysis on the class control flow graph. However, other dataflow analysis techniques could similarly be adapted.

We have demonstrated, by example, the ability of our technique to lead to construction of tests that uncover errors in a class. We are currently implementing a prototype dataflow tester for classes in C++, on which to perform experimentation. We are also investigating other issues related to dataflow testing for classes, such as techniques for providing partial dataflow analysis on classes that will permit users of the class to avoid complete reanalysis during integration.

References

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

[2] L.A. Clarke, A. Podgurski, D. Richardson, and S. Zeil. A comparison of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318 1332, November 1989.

[3] P.G. Frankl and R.K. Doong. Case studies on testing object-oriented programs. In *Proceedings of the Fourth Symposium on Testing, Analysis, and Verification*, pages 165–177, October 1991.

[4] P.G. Frankl and S. Weiss. An experimental comparison of the effectivenss of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[5] P.G. Frankl, S. Weiss, and E.J. Weyuker. ASSET: A system to select and evaluate tests. In *Proceedings of the IEEE Conference on Software Tools*, pages 72–79, April 1985.

[6] P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1493–1498, October 1988.

[7] M.J. Harrold and P. Kolte. Combat: A compiler based data flow testing system. In *Proceedings of the Pacific Northwest Quality Conference*, pages 311–323, October 1992.

[8] M.J. Harrold, J.D. McGregor, and K.J. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 68–80, May 1992.

[9] M.J. Harrold and M.L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.

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

[11] D. Hoffman and P. Strooper. Graph-based class testing. In *Proceedings of the 7th Australian Software Engineering Conference*, September 1993.

[12] D. Hoffman and P. Strooper. Graph-based module testing. In *Proceedings of the 16th Australian Computer Science Conference*, pages 479 487, February 1993.

[13] J. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the Fourth Symposium on Testing, Analysis, and Verification*, pages 87–97, October 1991.

[14] M. Hutchins, H. Foster, T. Goradia, and T.J. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, May 1994.

[15] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[16] S. Ntafos. An evaluation of the required element testing strategies. In *Proceedings of the 7th International Conference on Software Engineering*, pages 250 256, March 1984.

[17] T.J. Ostrand and E.J. Weyuker. Data flow-based adequacy analysis for languages with pointers. In *Proceedings of the Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 74–86, October 1991.

[18] 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.

[19] A.S Parrish, R.B. Borie, and D.W. Cordes. Automated flow graph-based testing of object-oriented software modules. *Journal of Systems Software*, 23:95–109, November 1993.

[20] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[21] C.D. Turner and D.J. Robson. The state-based testing of object-oriented programs. In *Proceedings of the Conference on Software Maintenance, 1993*, pages 302–310, September 1993.

[22] E.J. Weyuker. Fault detection using data flow testing or you showed me it was cheap, but is it any good? In *Proceedings of the Eighth Annual Northwest Software Quality Conference*, pages 213–217, October 1990.

[23] E.J. Weyuker. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering*, SE-16(2):121–128, February 1990.

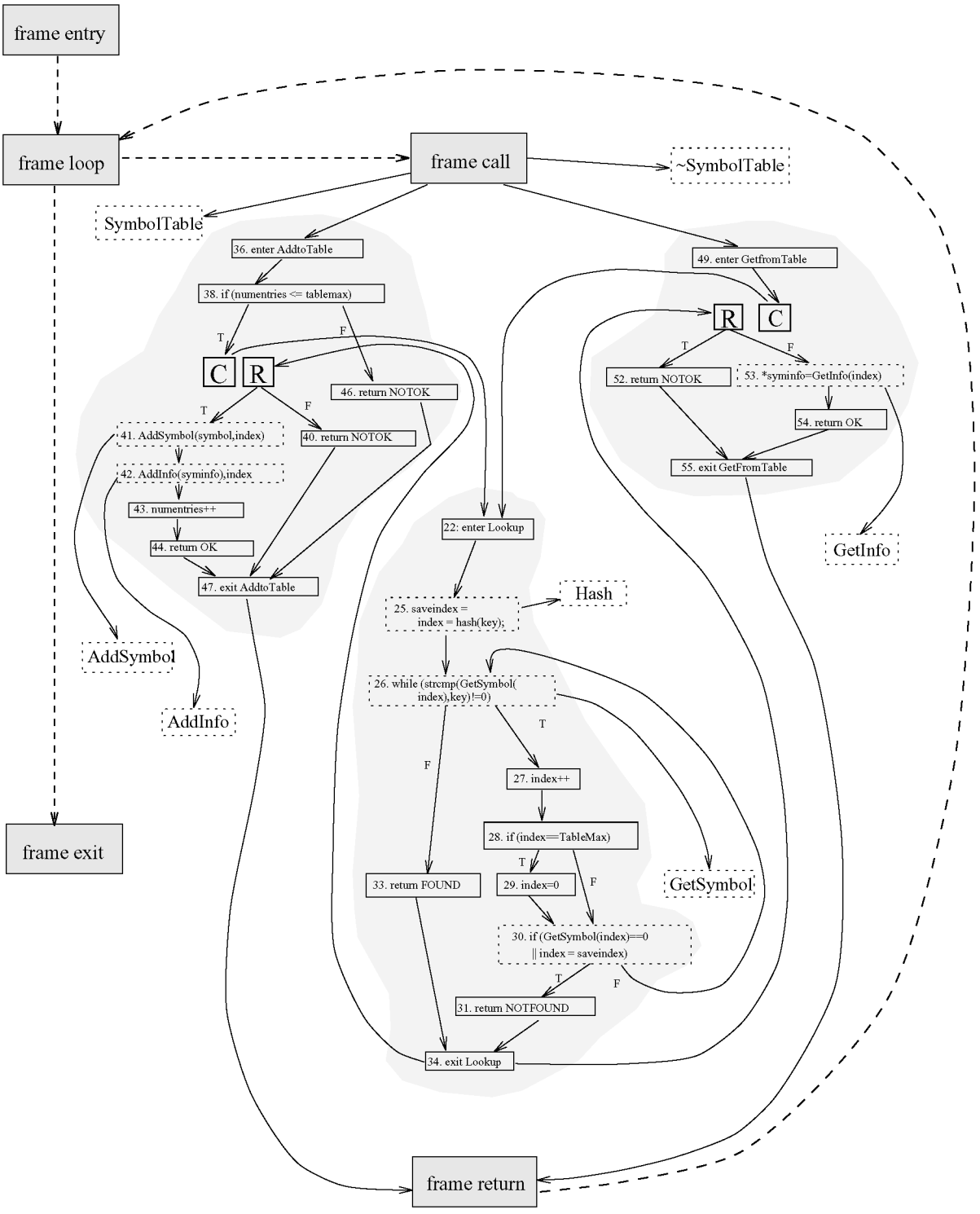

Figure 5: Partial CCFG for the `SymbolTable` class. The large shaded areas outline (clockwise from upper left) the control flow graphs for the `AddtoTable`, `GetfromTable`, and `Lookup` methods, respectively. Other methods are not expanded. Dotted boxes outline portions of the graph that have not been expanded.