Simultaneous Demand-Driven Data-flow and Call Graph Analysis

Gagan Agrawal
Department of Computer and Information Sciences
University of Delaware
Newark DE 19716
(302)-831-2783
agrawal@cis.udel.edu

Abstract

Recently, there has been much interest in performing demand-driven data-flow analysis in software development environments. Demand-driven analysis techniques compute data-flow fact(s) for a particular program point, rather than computing data-flow information for the entire program. The existing work in this area, however, assumes that the static call graph *is available for the entire program. Constructing exhaustive call graphs can be extremely time and space inefficient for large object-oriented programs. Therefore, it is desirable to compute the call graph information on a demand basis as well. In this paper, we present an algorithm for demand-driven construction of reaching definitions, which also performs call graph analysis on a demand basis.*

1 Introduction

There has been a significant interest in using demand-driven algorithms for data-flow analysis in software development environments [9, 14]. Demand-driven algorithms alleviate the need for expensive computation of data-flow information for the entire program. For example, during regression testing or integration level testing of a code [22], the set of reaching definitions may be required only for a particular variable at a particular point. Such information can be computed efficiently through demand-driven algorithms.

Previous demand-driven techniques have concentrated on languages like C and Fortran. In performing demand-driven analysis on a program, these techniques assume complete and accurate *call graph* information, i.e., which procedure(s) may be invoked at which call-site. However, constructing a call graph is often extremely space and time expensive for object-oriented languages. The experiments conducted by Chambers *et al.* at University of Washington have shown that for several Java programs comprising 7,000 to 30,000 lines of code, context-sensitive call graph construction either took more than 450 MB of space or did not finish execution within 24 hours [7, 11]. If program analysis is not used frequently as part of the software development environment, call graphs will not be computed and maintained. Therefore, in performing demand data-flow analysis, it is desirable to also construct the call graph in a demand-driven fashion.

In this paper, we present a demand-driven algorithm for computing the set of reaching definitions at a program point. In computing the set of reaching definitions, this algorithm does not assume an existing call graph. Instead, call graph analysis is also performed on a demand basis. Only the call-sites that can potentially propagate reaching definitions for a program point are analyzed for determining the set of procedures that can be invoked there.

Our technique for performing such demand-driven analysis will be useful in a number of software engineering tasks, like regression testing, integration testing and static assertion checking. Also, the simultaneous reaching definition and call graph analysis presented here can also be generalized for performing other data-flow analyses with demand call graph analysis.

The rest of the paper is organized as follows. We define the problem we are addressing and motivate the solution in Section 2. The representation Interprocedural Flow Graph used for presenting our analysis is explained in Section 3. Actual analysis is presented in Section 4. We comment on several aspects of our work and compare it with related work in Section 5 and conclude in Section 6.

2 Problem Definition

In this section, we first describe the language features our solution targets. We then state the main challenges in performing demand-driven analysis for reaching definitions and call graph construction.

2.1 Language Model

We are interested in treating common object-oriented languages like C++ and Java. At the same time, we are interested in focusing on a set of simple language features, so that we can present the details of our technique with simplicity.

A class has members fields and member methods or procedures. We assume that there are no global variables in the program. A member procedure of a class is denoted as `pclass::pname`, where `pclass` is the class in which this procedure is defined and `pname` is the declared name of this procedure. Besides scalar variables, there are variable that are references or pointers to an object of a declared class (which we will refer to as the *object references*). A base class can be extended by another class. In such a case, the base class is called a *superclass* and the class extending the base class is called a *subclass*. A class can only extend one superclass, treating languages with multiple inheritance is beyond the scope of this paper. The *set of subclasses* of a given class c is constructed transitively by including the class c, and any class that extends c or any class already included in the set. Similarly, the *set of superclasses* of a given class c is constructed transitively by including the class c, the class that the class c extended or any class that is extended by a class already included in the set.

We assume a statically-typed language in which a new object reference is created by a statement of the type `r = new Classname;` . Such a statement gives the type `Classname` to the variable `r`. The type of this instance of the variable `r` can only change as a result of another `new` statement.

The procedure calls are made as *static procedure calls*, in which an explicitly declared procedure is directly called by its name, or as *dynamic procedure calls*. A dynamic procedure call is of the format `r->pname(paramlist)`. We assume that the actual procedure invoked at this call-site depends only upon the type of `r`, and not on the type of arguments in the `paramlist`.

If r is of type `Classname`, then the procedure invoked at this call-site is the procedure `pname` defined in the class `Classname` (if it is defined there) or the procedure with the name `pname` declared in the nearest superclass of `Classname`. However, the type of the object that `r` is a reference to is usually not known through static declarations. If the static declaration of `r` is a reference to an object of the class `Declname`, `r` can be dynamically assigned to an object of any subclass of `Declname`.

We further assume that all parameter passing is by reference.

2.2 Motivating Example

To motivate the problem, consider the code shown in Figure 1. We assume that all parameters are passed by reference and all functions are declared to be virtual (i.e. can be over-written by a derived class). All member methods and fields are assumed to be public for simplicity. The class A is the base class, from which the class B is derived. Functions P, Q and R are each defined in both these classes.

Suppose we are interested in computing the set of reaching definitions for the variable `y` at the program point `s3`. The variable `y` is passed by reference at the call-site `cs1`. Thus, the reaching definition of `y` at the program point `s3` depends upon the actual procedure that may be invoked at the call-site `cs1`. However, the actual procedure invoked at the call-site `cs1`, in turn, depends upon the type of the object reference `x`. If `x` refers to object of type `A`, then the function `A::Q` will be invoked, whereas if `x` refers to object of type `B`, then the function `B::Q` will be invoked.

The type of the variable `x` at the program point `cs1`, in turn depends upon the type passed as a parameter when the function `A::P` is invoked. For determining this, we need to know at which program points can the function `A::P` be invoked, and what are the types of the parameter `x` at these call-sites. In the absence of the call graph, it is not known where the procedure `A::P` can be invoked. Looking at the class hierarchy, we know that this procedure can potentially be invoked at the call-sites `cs3` and `cs4`. Further backward analysis at the program points `cs3` and `cs4` can determine that the procedure `A::Q` can only be invoked at the call-site `cs4`. Further, the type of the parameter `x` that can be passed at `cs4` is the reference to class A. As a result, the procedure actually invoked

```
Class A {
        void P(A *x, int y);
        void Q(int y);
        void R() ;
}
Class B: public A {
        void P(A *x, int y) ;
        void Q(int y) ;
        void R() ;
}
A::Q(int y) {
        y += 1 ;                          s1
}
B::Q(int y) {
        y += 2 ;                          s2
}
```

```
A::P(A *x, int y) {
     x.Q(y) ;                            cs1
     write(y) ;                          s3
}
main() {
A* a;
A* b;
int i;
     ...                                 s4
     a.R() ;                             cs2
     a = new B ;                         s5
     b = new A ;                         s6
     a.P(b,i) ;                          cs3
     a = new A ;                         s7
     i = 1 ;                             s8
     a.P(a,i) ;                          cs4
}
```

Figure 1. Object-oriented program used as a running example in this paper. Definitions of functions A::R, B::R and B::P are not provided here.

at the call-site `cs1` is `A::Q`. In turn, we can determine that the reaching definition of the variable `y` at the program point `s3` arises from the statement `s1`.

An alternative approach to performing this analysis will be construct a call graph for the entire program [7, 11] and then apply existing techniques for demand-driven data-flow analysis [9, 14]. The obvious disadvantage of this approach will be that the call graph for the entire program will need to be constructed and maintained. In this code, the call graph can be constructed with relative ease by simple propagation of types. However, in general this has been shown to be a very time and space consuming process. If only a small number of call-sites may need to be resolved in determining the set of reaching types, then exhaustive call graph analysis will only incur unnecessarily high expenses. Therefore, in performing demand-driven analysis, it is useful to perform call graph analysis on a demand-driven basis as well.

3 Interprocedural Flow Graph

We use the interprocedural representation Interprocedural Flow Graph (IFG) initially proposed by Harrold and Soffa [13]. This representation is also closely based upon the representation Program Summary Graph (PSG) proposed by Callahan [5]. We initially give the original definition of IFG and then describe several extensions of IFG that we use for our purpose.

A Interprocedural Flow Graph (IFG) is a representation of the complete program that is much concise as compared to the Myers' Supergraph or Interprocedural Control Flow Graph (ICFG) [16, 17], but is more detailed than a call graph, and allows *flow-sensitive* interprocedural analysis.

Data-flow within procedures and between procedures is represented through edges between nodes of the following four types:

- *Entry nodes*; there is one entry node for each formal parameter of each procedure.
- *Exit nodes*; there is one exit node for each formal parameter of each procedure.
- *Call nodes*; there is one call node for each actual parameter at each call-site.
- *Return nodes*; there is one return node for each actual parameter at each call-site.

Edges between these nodes can be classified as *intraprocedural* and *interprocedural*. Intraprocedural edges summarize data-flow within the procedures.

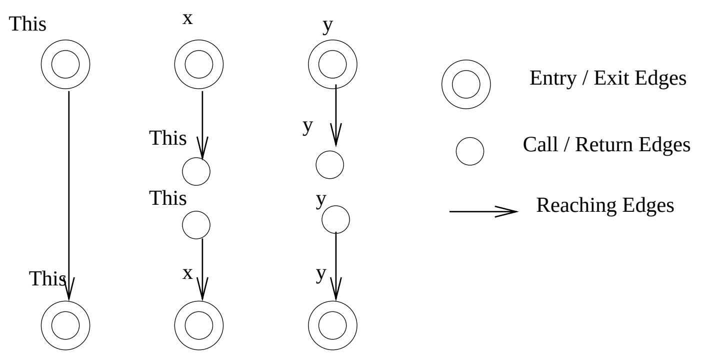

Figure 2. Procedure `A::P`'s portion of IFG

These edges are inserted after solving the standard data-flow problem of reaching definitions within each procedure [1]. Specifically, the intraprocedural edges are inserted:

- From an entry node to a call node if the value of the corresponding formal parameter at the procedure entry reaches the corresponding actual parameter at the call-site.
- From an entry node to an exit node if the value of the corresponding formal parameter at the procedure entry reaches a procedure return statement.
- From a return node to a call node if the value of the corresponding actual parameter at the call return reaches the corresponding actual parameter at the call-site.
- From a return node to an exit node if the value of the corresponding actual parameter at call return reaches a procedure return statement.

Interprocedural edges in the graph represent bindings of actual parameters to the formal parameters and vice-versa. Specifically, interprocedural edges are inserted:

- From a call node to an entry node to represent the binding of an actual parameter at the call-site to the formal parameter at procedure entry.
- From an exit node to a return node to represent the binding of a formal parameter at the procedure exit to the actual parameter at the call return.

Procedure `A::P`'s portion of IFG is shown in Figure 2.

Consider any call or exit node. The definition of the parameter this node represents may be generated in another procedure, in which case this node will have an entry node or a return node as a predecessor. Alternatively, the reaching definition may be generated within this procedure. To model this, we associate a set of reaching definitions with each call or exit node in the procedure. This set is denoted by LREACH, which stands for local reaching definitions. If a call or exit node denotes an object reference, we compute the possible reaching types of the object reference resulting from object creating statements within this procedure. Such a set of locally reaching types is denoted by LTYPES.

To perform demand-driven call graph analysis, we make the following two extensions to the original definition and construction method of IFG.

This pointer as a parameter: Consider a call site of the form `r->pname(paramlist)`. For our analysis, besides having call and return nodes corresponding to each actual parameter in the list `paramlist`, we also need to have a call and return node for the object reference `r`. We refer to such nodes as nodes for THIS pointer, consistent with the C++ terminology. We also insert one THIS pointer node each at procedure entry and return for each procedure.

Demand-Driven Construction: The Interprocedural Flow Graph is not fully constructed at the beginning of the analysis, since it will require performing reaching definition analysis on Control Flow Graphs of all procedures in the program, which can be very expensive and will defeat the purpose of doing demand-driven analysis. Instead, the program summary graph is constructed on a demand basis. If v is the call node for which we are determining the types, we perform a reachability analysis on the portion of the graph constructed and check if v is reachable from one of the nodes of a procedure p. If so, the CFG of the procedure p is constructed, and intraprocedural edges of the procedure p's portion of IFG are inserted. The portion of IFG constructed in performing the propagation analysis is referred to as the Partial Interprocedural Flow Graph (PIFG) and set of nodes in the PIFG is referred to as the *set of influencing nodes*.

4 Demand-Driven Analysis

In this section, we describe various phases of our algorithm for computing reaching definitions and call graph information on a demand basis.

4.1 Overview of the Algorithm

Given a program point and a variable, we need to compute the set of reaching definitions for this variable at this program point. If all the reaching definitions are assigned locally, the problem is trivial. Otherwise, we can relate the problem of computing the reaching definitions at this program point to one of computing the set of reaching definitions at a set of entry and return nodes. For example, in the code shown in Figure 1, the set of reaching definitions for the variable `y` at the program point `s3` is the same as the set of reaching definitions for the return parameter corresponding to `y` at the call-site `cs1`. We denote by V the set of such return and entry nodes. Once the set of reaching definitions has been computed for the nodes in the set V, answering the original question is straight-forward.

The most important goal for our algorithm is to analyze as few procedures as possible while determining the set of reaching definitions for the set of nodes V. Our algorithm initially assumes a sound or conservative call graph, i.e., one with much larger number of edges than what can be taken during actual executions. This initial sound call graph is constructed using the results of Class Hierarchy Analysis (CHA) [6]. Such an initial sound call graph is also not constructed explicitly for the entire program, but is constructed on a demand basis. Each procedure's components are added only after it is known that this procedure may influence the the set of reaching definitions for the nodes in the set V.

There are two main phases in our algorithm. Initially, we perform reachability analysis using the sound call graph to determine the set of influencing nodes and to construct the Partial Interprocedural Flow Graph (PIFG). The second phase involves performing data-flow propagation on the PIFG to improve the precision of the call graph using reaching types information. In the process, we also refine the reaching types information.

Our algorithm is presented in the next four subsections. In Section 4.2, we describe the technique for construction of initial call graph. We then introduce a set of definitions for presenting our work in Section 4.3. In Section 4.4, we describe the reachability analysis for computing the set of influencing nodes and constructing the PIFG. Finally, we describe the technique for improving the precision of the sound call graph and the precision of the reaching types information by performing propagation over the set of influencing nodes in Section 4.5.

4.2 Initial Conservative Call Graph

We can construct a relatively accurate initial call graph by performing inexpensive Class Hierarchy Analysis (CHA) [6].

CHA involves having knowledge of all the classes declared in the entire program, including which class extends another class, and the set of procedures declared in each of the classes. Consider a call-site of the form `r.rname()`, such that the declared type of `r` is `rclass`. Let $\mathcal{R}$ be the set of subclasses of `rclass`. For each class in the set $\mathcal{R}$, we determine the nearest superclass (including itself) in which a procedure with the name `rname` is declared. Let us denote such a set of classes by $\mathcal{R}'$. Then, as a result of class hierarchy analysis, we know that the possible procedures that can be called at this call-site are of the form `pclass::rname`, where `pclass` belongs to the set $\mathcal{R}'$.

Alternatively, consider a procedure p of the form `pclass::pname`. By knowing all class declarations in the program, we can determine the set of subclasses of `pclass`. By further examining the procedures declared in each of these classes, we can narrow this set down to classes for which `pclass` is the earliest superclass for which the procedure `pname` is defined. Let us denote such a set by $\mathcal{S}$. The procedure p can be called at any dynamic call-site of the form `r.pname` where the declared type of the reference `r` belongs to the set $\mathcal{S}$.

```
Let $V$ be the initial set of nodes
Let $p$ be the procedure to which nodes in $V$ belong
Initialize $Workset$ to $\{V\}$
Initialize $Procset$ to $\{p\}$
Initialize all nodes to be not *marked*
$Construct\_IFG\_Portion(p)$

*While* $Workset$ is not empty
    Select and remove vertex $u$ from $Workset$
    **case (type of u):**
        **call or exit:**
            *foreach* predecessor $w$ of $u$
                *If* $w$ is not *marked*
                    $Workset = Workset \cup \{w\}$
        **return:**
            *If* THIS_NODE$(u)$ is not *marked*
                $Workset = Workset \cup$ THIS_NODE$(u)$
            *foreach* possibly called function $q$
                *If* $q \notin Procset$
                    $Procset = Procset \cup \{q\}$
                    $Construct\_IFG\_Portion(q)$
            *foreach* predecessor $w$ of $u$
                *If* $w$ is not *marked*
                    $Workset = Workset \cup \{w\}$
        **entry:**
            $Workset = Workset \cup$ THIS_NODE$(u)$
            *foreach* possible callee function $q$
                *If* $q \notin Procset$
                    $Procset = Procset \cup \{q\}$
                    $Construct\_IFG\_Portion(q)$
            *foreach* predecessor $w$ of $u$
                *If* $w$ is not *marked*
                    $Workset = Workset \cup \{w\}$
```

Figure 3. Constructing the Partial Interprocedural Flow Graph (PIFG)

4.3 Preliminary Definitions

In presenting our technique, we use the following definitions. We had earlier defined LREACH and LTYPES, the set of locally reaching definitions and locally reaching types.

$pred(v)$: The set of predecessors of the node v in the PIFG. This set is initially defined during the construction of PIFG and is not modified as the type information becomes more precise.

$proc(v)$: This relation is only defined if the node v is an entry node or an exit node. It denotes the name of the procedure to which this node belongs.

TYPES(v): The set of types associated with a node v in the PIFG during any stage in the analysis. This set is initially constructed using Class Hierarchy Analysis and intraprocedural propagation, and is later refined through data-flow propagation.

RDEFS(v): The set of reaching definitions for the node v.

THIS_NODE(v): This is the node corresponding to the THIS pointer at the procedure entry (if v is an entry node), procedure exit (if v is an exit node), procedure call (if v is a call node) or call return (if v is a return node).

THIS_TYPE(v): If the vertex v is a call node or a return node, THIS_TYPE(v) returns the types currently associated with the call node for the THIS pointer at this call-site. This relation is not defined if v is an entry or exit node.

PROCS(S): Let S be the set of types associated with a call node for a THIS pointer. Then, PROCS(S) is the set of procedures that can actually be invoked at this call-site. This function is computed using Class Hierarchy Analysis (CHA).

4.4 Constructing the Set of Influencing Nodes

We now describe how we compute the set of nodes in the PIFG for the entire program that influence the set of procedures invoked at the given call-site c_i. The PIFG for the entire program is never constructed. However, for ease in presenting the definition of the set of influencing nodes, we assume that the PIFG components of all procedures in the entire program are connected based upon the initial sound call graph.

Let V be the set of initial nodes in the IFG for which we are computing the set of reaching definitions.

- $(v \in V) \rightarrow v \in S$

- $(x \in S) \wedge (y \in pred(x)) \rightarrow y \in S$
- $x \in S \rightarrow$ THIS_NODE$(x) \in S$

Starting from any node v in the set V, we include the predecessors of any node already in the set, till we reach internal nodes that do not have any predecessors. For any node included in the set, we also include the corresponding node for the THIS pointer (denoted by THIS_NODE) in the set.

Such a set of influencing node and the partial PIFG can be constructed by an iterative algorithm, which is shown in Figure 3. Two main data-structures maintained in the algorithm are $Workset$ and $Procset$. $Workset$ is the set of nodes whose predecessors have not been analyzed yet. $Procset$ is the set of procedures that have been analyzed and whose portions of the PIFG has been constructed.

The algorithm progresses by removing a node from the $Workset$. If this node is a call or exit node, all the predecessors of this node are within the same procedure. These predecessors are added to the $Workset$. If the node u (removed from the $Workset$) is a return node, the predecessors of this node are the exit nodes of the procedures that can be invoked at this call-site. Such a set of procedures is known (not necessarily accurately) from our construction of the initial sound call graph. Let q be any such procedure. If q is not in the set $Procset$ (i.e. it has not been analyzed yet), then the function $Construct\_IFG\_Portion(q)$ is invoked. This function analyzes the CFG of the procedure q to construct its portion of the PIFG. For each callee of q that has been analyzed, edges from its call nodes to entry nodes of q and edges from exit nodes of q to its return nodes are inserted. Similarly, for each function called by q that has been analyzed, we insert edges from call nodes at q to its entry nodes from its exit nodes to the return nodes at q. After all such procedures called at this call-site have been analyzed and edges have been inserted, we add the predecessors of the node u to the $Workset$.

The edges inserted at these call-sites are obviously based upon an initial sound call graph, that needs to be refined by our analysis. For this purpose, we need to know the types associated with the THIS_NODE at the call-site corresponding to the return node. For this reason, we also add THIS_NODE(u) to the $Workset$.

The actions taken for an entry node are very similar to the actions taken for a return node. The only difference is that instead of analyzing the procedures that can be called at that call-site, we analyze the procedures that have a call-site that can invoke this procedure.

The PIFG constructed for our example program is shown in Figure 4.

4.5 Refining Data-Flow Information

The next step in the algorithm is to perform iterative analysis over the set of nodes in the Partial Program Summary Graph (PIFG) to compute the set of types associated with the nodes and then compute the set of reaching definitions. The initial values of TYPES(*v*) are computed through class hierarchy analysis that we described earlier in this section. If a formal or actual parameter is declared to be a reference to class `cname`, then the actual runtime type of that parameter can be any of the subclasses (including itself) of `cname`. Also, the initial value of RDEFS is the universal set.

The refinement stage for TYPES can be described by a single equation, which is shown in Figure 5. Consider a node v in PIFG. Depending upon the type of v, three cases are possible in performing the update:

1. v is a call or exit node,
2. v is an entry node, and
3. v is a return node.

In Case 1., the predecessors of the node v are the internal nodes, the entry nodes for the same procedure, or the return nodes at one of the call-sites within this procedure. The important observation is that such a set of predecessors does not change as the type information is made more precise. So, the set TYPES(v) is updated by taking union of LTYPES(v) with the union of the sets TYPES(p) over the predecessors of the node v.

We next consider case 2, i.e., when the node v is an entry node. $proc(v)$ is the procedure to which the node v belongs. The predecessors of such a node are call nodes at all call-sites at which the function $proc(v)$ can possibly be called, as per the initial call graph assumed by performing class hierarchy analysis and intraprocedural propagation. Such a set of possible call-sites for $proc(v)$ gets restricted as interprocedural type propagation is performed. Let p be a call node that is a predecessor of v. We want to use the set TYPES(p) in updating TYPES(v) only if the call-site corresponding to p invokes $proc(v)$. We determine this by checking the condition $proc(v) \in$ PROCS(THIS_TYPE(p)). The function THIS_TYPE(p) determines the types

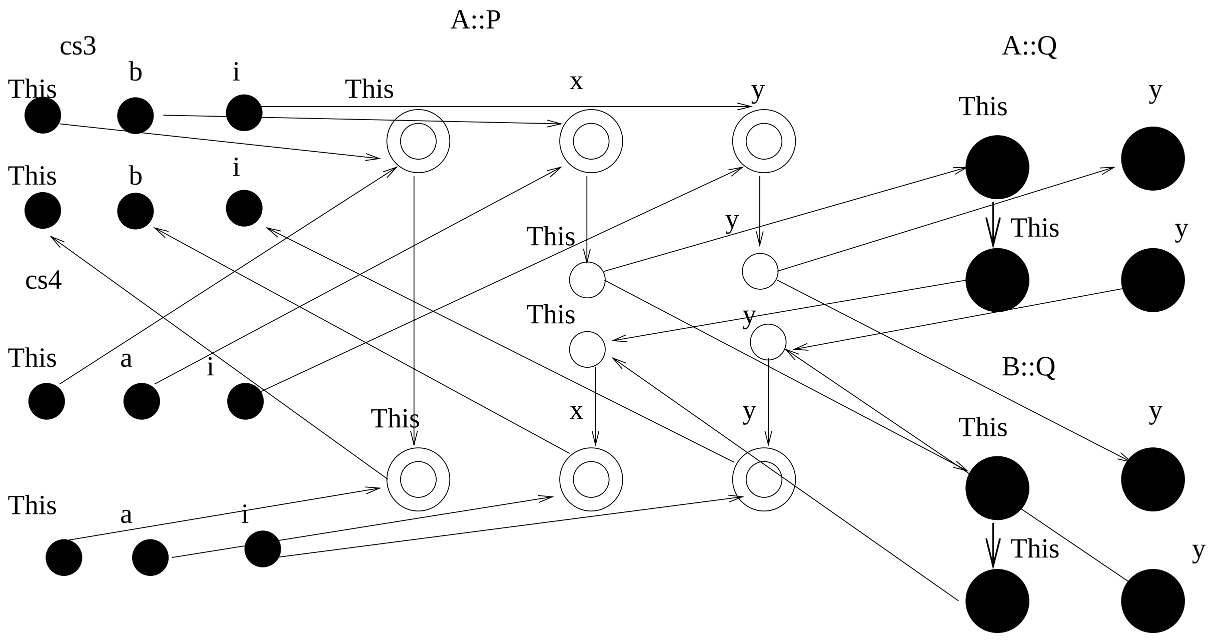

Figure 4. Partial Interprocedural Flow Graph Constructed for the Example Program

$$\text{TYPES}(v) \;=\; \begin{cases} \text{LTYPES}(v) \bigcup (\bigcup_{p \,\in\, pred(v)} \;\text{TYPES}(p)\,) & \text{if v is call or exit node} \\ \bigcup_{(p \,\in\, pred(v)) \,\wedge\, (proc(v) \,\in\, \text{PROCS}(\text{THIS\_TYPE}(p)))} \;\text{TYPES}(p) & \text{if v is an entry node} \\ \bigcup_{(p \,\in\, pred(v)) \,\wedge\, (proc(p) \,\in\, \text{PROCS}(\text{THIS\_TYPE}(v)))} \;\text{TYPES}(p)\;) & \text{if v is a return node} \end{cases}$$

$$\text{RDEFS}(v) \;=\; \begin{cases} \text{LREACH}(v) \bigcup (\bigcup_{p \,\in\, pred(v)} \;\text{RDEFS}(p)\,) & \text{if v is call or exit node} \\ \bigcup_{(p \,\in\, pred(v)) \,\wedge\, (proc(v) \,\in\, \text{PROCS}(\text{THIS\_TYPE}(p)))} \;\text{RDEFS}(p) & \text{if v is an entry node} \\ \bigcup_{(p \,\in\, pred(v)) \,\wedge\, (proc(p) \,\in\, \text{PROCS}(\text{THIS\_TYPE}(v)))} \;\text{RDEFS}(p)\;) & \text{if v is a return node} \end{cases}$$

Figure 5. Data-Flow Equation for Propagating Type and Reaching Definition Information

currently associated with the THIS pointer at the call-site corresponding to p and the function PROCS determines the set of procedures that can be called at this call-site based upon this type information.

Case 3 is very similar to the case 2. If the node v is a return node, the predecessor node p to v is an exit node. We want to use the set TYPES(p) in updating TYPES(v) only if the call-site corresponding to v can invoke the function $proc(p)$. We determine this by checking the condition $proc(p) \in$ PROCS(THIS_TYPE(v)). The function THIS_TYPE(v) determines the types currently associated with the THIS pointer at the call-site corresponding to v and the function PROCS determines the set of procedures that can be called at this call-site based upon this type information.

The equation for computing the set of reaching definitions is analogous and is shown the same figure. As the type information becomes more refined, more refined reaching definition information can be computed.

5 Discussion and Related Work

We now explain some of the limitations of our approach and the possibilities for future work. In performing interprocedural propagation, we have not addressed the problem of *preserving calling context*. If a procedure is invoked at multiple call-sites, a flow-sensitive interprocedural representation like IFG has *invalid* or *unrealizable* paths in it. Propagating data-flow information along these paths can lead to inaccuracies. This limitation applies both to our propagation of types information and the reaching definitions information. Another inaccuracy can come in because of the way we initialize the data-flow values. We start with an optimistic estimate and then improve upon it. In performing exhaustive analysis, we would have preferred to start with a conservative solution and then add further elements. However, this is not feasibly while performing demand analysis, because of the way the initial conservative call graph is constructed.

The previous work in the area of call graph construction [2, 3, 4, 7, 8, 10, 11, 12, 15, 19, 18, 20, 21, 23, 25, 24, 26] has only focussed on exhaustive analysis and has not considered demand-driven analysis. Similarly, the previous work in the area of demand-driven data-flow analysis [9, 14] has assumed that a complete call graph has already been constructed before initiating the demand-driven analysis.

6 Conclusions

Demand-driven analysis techniques have been found useful in software engineering environments, for applications like regression testing, integration testing and static assertion checking. An important limitation of the existing techniques for demand-driven analysis has been to assume a static call graph. In this paper, we have presented a novel demand-driven analysis technique which performs call graph and reaching definition analysis simultaneously. Only the call sites that can potentially influence the accuracy of the reaching definition information being computed are analyzed. Our technique can significantly ease computation of data-flow information for object-oriented programs, for software engineering tasks like regression testing, integration testing and static assertion checking.

Acknowledgments

This research was supported by NSF CAREER award ACI-9733520.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] David Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pages 324–341, October 1996.

[3] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[4] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Portland, Oregon, January 1994.

[5] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988.

[6] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 93–102, La Jolla, California, 18–21 June 1995. *SIGPLAN Notices* 30(6), June 1995.

[7] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings of the POPL'98 Conference*, 1998.

[8] Amer Diwan, J. Elliot Moss, and K. Mckinley. Simple and effective analysis of statically typed object-oriented programs. In *Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pages 292–305, October 1996.

[9] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedual data flow. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, California, January 1995.

[10] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural Points-to analysis in the presence of function pointers. *SIGPLAN Notices*, 29(6):242–256, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[11] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1997.

[12] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[13] 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.

[14] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *In SIGSOFT '95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, 1995.

[15] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284, Charleston, South Carolina, January 1993.

[16] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

[17] E. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth ACM Symposium on the Principles of Programming Languages*, pages 219–230, January 1981.

[18] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 367–378, San Francisco, California, January 1995.

[19] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, pages 146–161, November 1991. Published as Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, number 11.

[20] Hemant Pande and Barbara Ryder. Data-flow-based virtual function resolution. In *Proceedings of the Third International Static Analysis Symposium*, 1996.

[21] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, pages 324–340, October 1994.

[22] Gregg Rothermel and M. J. Harrold. Analyzing regression test selection. *IEEE Transactions on Software Engineering*, 1996.

[23] B. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[24] O. Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 190–198, New Haven, CN, June 1991.

[25] Olin Shivers. Control-flow analysis in Scheme. *SIGPLAN Notices*, 23(7):164–174, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[26] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Las Vegas, Nevada, January 1980.