A Framework for Partial Data Flow Analysis*

Rajiv Gupta and Mary Lou Soffa
{gupta,soffa}@cs.pitt.edu
Department of Computer Science
University of Pittsburgh
Pittsburgh, Pa, 15260

Abstract

Although data flow analysis was first developed for use in compilers, its usefulness is now recognized in many software tools. Because of its compiler origins, the computation of data flow for software tools is based on the traditional exhaustive data flow framework. However, although this framework is useful for computing data flow for compilers, it is not the most appropriate for software tools, particularly those used in the maintenance stage. In maintenance, testing and debugging is typically performed in response to program changes. As such, the data flow required is demand driven from the changed program points. Rather than compute the data flow exhaustively using the traditional data flow framework, we present a framework for partial analysis. The framework includes a specification language enabling the specification of the demand driven data flow desired by a user. From the specification, a partial analysis algorithm is automatically generated using an L-attributed definition for the grammar of the specification language. A specification of a demand driven data flow problem expresses characteristics that define the kind of traversal needed in the partial analysis and the type of dependencies to be captured. The partial analyses algorithms are efficient in that only as much of the program is analyzed as actually needed, thus reducing the time and space requirements over exhaustively computing the data flow information. The algorithms are shown to be useful when debugging and testing programs during maintenance.

Keywords - *control flow graph (CFG), program debugging, program testing, code optimization.*

1 Introduction

Static program analysis was first developed in the early 70s for use in compiler optimizations, recognizing that knowledge about the flow of data values in a program leads to better register allocation and more run-time efficient code. Its use in parallelizing compilers is invaluable, as code must be transformed using data dependency information in order to fully exploit the parallel architectures [5, 17]. In addition, static analysis has also become a primary component of many software tools, such as editors [20], debuggers [25], software testers [3, 9, 19] program integration [10], and parallel program analyzers [2, 4]. Data flow has been proven to be especially useful in tools for the maintenance stage [6, 8]. Although compilers and software tools utilize static analysis to improve their capabilities and performances, there are important differences in the data flow information needed between these two classes of software.

Compilers require information about the flow of data for an entire program, as global optimizations are typically applicable to all code in the program. As such, data flow information is computed exhaustively using the traditional data flow framework [12] and is computed before optimizations are applied. The types of data flow or data dependency information needed are based on the kinds of optimizations and parallelizing transformations to be applied and are thus known beforehand. And lastly, optimizations are applied in many cases after a program has been debugged. Thus, the data flow computation is not really designed to easily incorporate changed program text.

On the other hand, data flow needed by many software tools is demand driven from one or more program points rather than exhaustive. For example, when debugging, we may want to know what data values reach a use at a program point or what statements impact on the value of a variable at a program point. In data flow testing, after a change has been made in a program, we want to know the impact that change has on the data values that it can reach. Thus, software tools are interested in the flow of data from program points. Also, the data flow problems to be solved are not fixed before the software tool executes but can vary depending on the user. For example, at a program point during debugging, a user may want to ask such questions as will a value reach a point along any path and what value must reach a point along all paths, as well as other questions that would help locate bugs. And lastly, many tools are used while the program is under development or maintenance and thus changes in the program are expected and must be efficiently handled.

Thus, exhaustive data flow information is needed in compilers whereas the data flow needed in a number of software tools is demand driven. The types of data flow needed is fixed for compilers but not for software tools, and the software tools need to efficiently respond to program changes. Although these basic differences

*Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-9109089 to the University of Pittsburgh.

exist in the data flow requirements for compilers and software tools, exhaustive algorithms derived from the data flow framework are typically used to compute the data flow for software tools. This approach causes the computation of data flow information about parts of a program that is not required by the data flow problem. When changes are made to code, the data flow has to be recomputed exhaustively and compared to previous data flow or has to be incrementally updated, under the assumption that exhaustive data flow has already been computed [1, 18, 22]. A major problem with computing data flow information exhaustively is the high cost both in execution time and memory demands. Experimental studies show that performing analyses even over small or medium size programs can take several hours [13].

In order to provide more *flexibility* and *efficiency* in the data flow computation for software tools, we present a framework for the computation of demand driven data flow using partial analysis algorithms. As this framework supports the computation of demand driven data flow from a program point, only the part of the program required for analysis is used to compute the data flow information. The framework is general in that many types of demand driven data flow problems, needed for software tools, can be expressed and computed. And lastly, a specification technique is included with the framework that enables the specification of data flow problems and the automatic generation of algorithms to perform the partial analysis. With this facility, the user is provided with a model for data flow problems and can express the particular problem of interest in the specification language. Using our framework, whenever data flow information with particular characteristics is required, the user only has to write a short specification identifying these characteristics. The characteristics identify the type of traversal through the program that is needed in the analysis and the dependencies required. Besides the specification technique, the framework also contains an L-attributed definition of the grammar for the specification language to actually generate the appropriate partial analysis algorithms. The framework is flexible in that additional characteristics of data flow problems can be easily added. We demonstrate the framework for a set of characteristics derived from common data flow problems for software tools. A prototype has been implemented and the utility and efficiency of the partial analysis algorithms in testing and debugging is examined.

A partial analysis algorithm produced by our technique is efficient in that the analysis is controlled by the dependencies being sought. Only nodes that must be visited to compute the required dependencies are visited, i.e., the complexity grows with size of the (partial) solution that is computed. Thus, we are computing less data flow information which results in both space and time efficiencies. Our algorithms use the control flow graph as the program representation. Another type of representation that has been used to compute static program slice, a type of demand driven data flow, is the program dependence graph [5, 11]. However, this representation needs to have the data flow computed exhaustively and then selects the information to present to the user, using the program dependence graph. Also, slices using the program dependence graph are defined only from program points where values are used; for example, in debugging, more flexibility is needed for we may want a slice from a program point where a variable is not used.

The next section of the paper discusses the characteristics and specification that we include in our framework for demand driven data flow for software tools. The technique for the automatic generation of partial analysis algorithms is presented in section 3. Section 4 demonstrates our technique through the specification of various partial data flow algorithms useful in debugging, testing, and test case generation. Section 5 considers the related work and a discussion of an implementation is included in section 6.

2 Characteristics and Specification of Demand Driven Data Flow

We begin by identifying general properties of demand driven data flow. Demand driven data flow is defined from a program point or set of points. The end of a program is a valid program point, indicating that data flow information is required about the entire program or all possible execution paths. Demand driven data flow captures the data dependencies relative to a program object, such as a set of variables or statements.

Assume that we want to know where a newly introduced definition of a variable at a program point may be used (i.e., partial reachable uses) in def-use testing of a program. In this case, a forward traversal must be made from a definition **searching** for the dependencies that can exist along any path from the given program point. Since interest is only in the dependence between a definition and its use, only direct dependencies are required. When a use of a variable is found, the statement is added to the data flow set and the search continues. The search for a use of a variable along a path ends when another definition of that variable is found. During the search, information reflecting the data flow information being sought is propagated. The identifying characteristics are that (i) the search is forward from a program point, (ii) uses are needed to identify dependencies, (iii) only immediate dependencies are required, and (iv) dependencies found along any path are needed.

Consider a different type of demand driven data flow problem, that of computing information useful during program debugging. Given a variable use, we want to know the locations of all definitions on which the use is directly or indirectly flow dependent. If any of these definitions are not constants, then another search must be established to determine their definitions, or the closure of the dependencies. Thus, the set of variables whose dependencies are required changes as data flow information is added to the computed data flow set. When a statement is added to the data flow set, other variables' dependencies must be found and these variables are added to the set of variable definitions being searched. Thus, variables whose data dependencies are required to produce the needed information are both propagated and spontaneously generated at a statement. The identifying characteristics are that (i) the search is backwards from a program point; (ii) definitions are needed to identify dependencies; (iii) the closure of dependencies is needed; and (iv) each path from the given program point specificed in the criterion is searched for dependencies.

The two types of data flow dependencies provided in our framework are *variable* dependencies that relate definitions and uses of variables and *statement* dependencies that compute structural relationships among the statements (e.g., dominators). To specify a starting point for a *variable* computation the user specifies a variable of interest at a program point. In case of *structural* dependencies the user specifies statements of interest at a program point.

DDConstruct → `Construct name` : *DDSpecify*
DDSpecify → *Variable* | *Statement*
Variable → `Vdep` *Search Reference*
Statement → `Sdep` *Search*
Search → *Direction Extent Path*
Direction → `forward` | `backward`
Extent → `immediate` | `closure`
Path → `all` | `any`
Reference → `def` | `use`

DDCompute → `Compute name` :- (*DDStart*)
DDStart → *VInput* | *SInput*
DDOutput → *VOutput* | *SOutput*
ProgPoint → `in` | `before` | `after`
VInput → variable *ProgPoint* statement
{ , variable *ProgPoint* statement }*
VOutput → { (variable *ProgPoint* statement, statement) }*
SInput → statement {, statement }*
SOutput → { (statement, statement) }*

Figure. 1: Specifying the Construction and Computation of Demand Driven Data Flow.

To specify demand driven data flow computations of different types we must specify the characteristics that describe the nature of the *search* that will enable the capture of relevant information by the data flow computation. In Fig. 1, we present the grammar for our specification language. The specification for the nature of search includes the *direction* in which the search is to take place. The search can be carried out in either the `forward` or `backward` direction from the program point specified by the user. The *extent* of the search indicates whether the search should terminate when all statements that have a direct relationship (`immediate`) with the criterion have been found or whether it should continue until all statements that have direct or indirect (`closure`) relationship with the criterion have been found. For a forward problem the user must indicate whether the statements being searched for are reachable from a given program point along `all` incoming edges of the statements or at least **any** one incoming edge. Similarly for a backward problem the user must specify whether from the statements being searched, a given program point can be reached along `all` outgoing edges of the statements or at least along **any** one outgoing edge. In the case of *variable* dependencies the user must specify whether the search is being carried out for definitions (`def`) of variables or uses (`use`) of variables.

The user first specifies a demand driven data flow problem using the *DDConstruct* specification given in Fig. 1. This specification associates a name with a partial analysis algorithm at the time the code is constructed. When a particular data flow computation is needed using the constructed partial analysis algorithm, the starting point is specified using the *DDCompute* specification. After the data flow has been computed, the name assigned to the computed set enables the user to access the items. When the `construct` statement shown below is encountered, an algorithm for computing definitions that reach uses of variables at various program points is constructed from the specification. The execution of the `compute` statement causes this algorithm to determine the reaching definitions for the given starting point in the program specified in `VInput`.

```
Construct ReachingDefs:
        Vdep backward immediate any def
Compute ReachingDefs :- ( VInput )
```

3 Generating Partial Analysis Algorithms

Next we describe the construction of a partial analysis algorithm from its specification. The construction of the algorithm is carried out during the parsing of the demand driven data flow specification. The actions required for constructing the algorithm are described by an L-attributed definition associated with the grammar in Fig. 1. An L-attributed grammar allows the use of synthesized attributes as well as restricted types of inherited attributes. The values of these attributes represent the characteristics of the data flow as well as the code for the partial data flow algorithm. Due to space consideration in this abstract, we make a simplifying assumption in the following discussion. Although array variables can be handled by our technique, we only discuss scalar variables. The primary modification for array variables is the terminating condition. In general, the search for array variables has to progress to the start of the program in a forward search or the end of a program in a backward search.

The nature of the search to be carried out during partial analysis is described by the synthesized attributes `Order`, `Prev`, `Next`, `Ext`, and `Meet` associated with the symbol *Search* (see Fig. 2). The attribute `Order` describes the order in which nodes are examined during the computation of data flow, the attributes `Next` and `Prev` identify the direction in which the flow graph is traversed during the computation of the data flow, `Ext` represents the scope of the search, and the meet operator specifies whether the information must be obtainable along **all** paths or at least

any one path. The `Order` is ***depth-first*** for **forward** data flow problems and is ***reverse depth-first*** for **backward** data flow problems. In the case of a **forward** search the `Prev` set of a statement is the set of the statement's predecessors (*Pred*) and the `Next` set is the statement's successors (*Succ*) in the control flow graph. In the case of a **backward** data flow problem it is the opposite. The `Meet` operator is ***union*** for **any** path problems and ***intersection*** for **all** path problems. An additional attribute `Point` is used during the construction of new criteria that are added during the computation of the demand driven data flow when the *Extent* of the search is specified as **closure**. Its value is *after* for forward problems and *before* for backward problems.

The properties of the data flow problem determined from this specification are described by the attributes `Found` and `New` of the symbol *Reference*. The attribute `Ext` is inherited by *Reference*. The `Found` attribute identifies the set associated with a statement which is examined to determine whether the statement should be included in the data flow set. Thus, it is the statement's reference set (*Ref*), if we are looking for uses of variables (**use**) and it is the statement's definition set (*Def*), if we are looking for the definitions of variables (**def**). The attribute `New` identifies the set specifying the additional information regarding a statement that the partial analysis algorithm must find after a statement has been included in the data flow set. Thus, it is used when the extent of the search is specified as **closure**. The extent of the search can be determined by examining the inherited attribute `Ext` associated with *Reference*. If we are looking for uses of a variable (**use**), then the `New` set is the *Def* set of the statement just included in the data flow set. However, if we are looking for definitions (**def**), then the `New` set is the *Ref* set of the statement included in the set.

Let us next consider the computation of variable dependencies. The code that computes data flow is constructed from the attributes of the data flow problem associated with the symbols *Search* and *Reference* (see Fig. 3). The resulting code is represented by the value of the attribute `Code` associated with *Variable*. In the code for the value of attribute `Code`, the sets Input and Output represent data flow sets associated with each statement maintained to assist in capturing data dependencies. These sets contain names of variables, and their associated program points, whose definitions/uses are being searched. In the case of a forward data flow problem, the Input set contains variables of interest at the point just before a statement and the Output set contains relevant variables after the statement. In the case of a backward data flow problem it is the opposite. The variables in the Output set of a statement consist of ***spontaneous*** variables (Soutput) and ***propagated*** variables (Poutput). The spontaneous variables are those variables which are included in the set due to the statement itself, that is, they are generated by the statement. Thus, the variables in the Soutput set of a statement s are those variables which are being considered due to the inclusion of s in the data flow set or because s is one of the statements that is specified as part of the criterion. The propagated variables are those variables that are simply propagated through the statement. Thus, the variables in the Poutput set are those that were also present in the Input set of the statement and were neither killed nor generated by the statement.

The statement n represents the statement currently being examined by the algorithm. This statement is obtained from the ***worklist*** of statements that is to be examined by the partial analysis algorithm. The statement n is examined for the definitions/uses of variables that meet the required dependencies. If no dependencies are found in a statement, the search must progress along the successors/predecessors of this statement. However, if the definition/use of a variable along a path is found that satisfies the dependency, or it is determined that none will be found along the current path, then the search for that variable along that path terminates. The search for other variables in the list, if any, continues. The continuation of the search along predecessor/successor statements is achieved by including these statements in the *worklist*. If the definitions/uses for all variables along n have been found (i.e., Output[n] $= \phi$), no new statements would be added to the *worklist*.

For statement dependencies, the data flow sets contain statements of interest to the user. The statements are propagated as far as they can be propagated under the user specified search characteristics. Unlike variable dependencies, definitions of variables cannot stop the propagation of statements since we are simply interested in structural relationships among the statements and not the data dependence relationships. In case of closure if a statement is included in the data flow set then it is also used to construct a new criteria (see Fig. 4).

Finally the code for capturing relevant information must be embedded in a loop, shown in Fig. 5, which examines the statements in the ***worklist*** one at a time. The ***worklist*** is initialized based upon the input program points of the criterion. At any given point in the algorithm the *worklist* indicates how far the search has progressed. All of the elements in this list are examined for the relevant information and the *worklist* is appropriately updated. If all relevant information along all paths being searched has been found, no new statements would be added to the *worklist* and it becomes empty. At this point the data flow set has been computed and the algorithm terminates. The statements in the *worklist* are ordered in depth-first/reverse-depth-first order and this ordering is maintained as new statements are added to the *worklist*. This improves the efficiency of the algorithm by reducing the number of times a statement is examined. The code that represents the partial analysis algorithm is found in the synthesized attribute `Code` of the symbol *DDSpecify*.

We only described the actions that construct the core of the partial analysis algorithm. Additional attributes that initialize the Input, Soutput, Poutput, and Output sets and the *worklist* at the the start of the algorithm are omitted.

```
Search→ Direction Extent Path [
      #Search.Order = #Direction.Order; #Search.Prev = #Direction.Prev;
      #Search.Next = #Direction.Next;   #Search.Ext = #Extent.Ext;
      #Search.Meet = #Path.Meet;        #Search.Point = #Direction.Point; ]

Direction→ forward [ #Direction.Order = "Depth-First"; #Direction.Prev = "Pred";
                     #Direction.Next = "Succ"; #Direction.Point = "after" ]
        | backward [ #Direction.Order = "Reverse-Depth-First"; #Direction.Prev = "Succ";
                     #Direction.Next = "Pred"; #Direction.Point = "before" ]

Extent → immediate [ #Extent.Ext = "Immediate" ] | closure [ #Extent.Ext = "Closure" ]

Path → all [ #Path.Meet = "∩" ] | any [ #Path.Meet = "∪" ]

Reference → def [ #Reference.Found = "Def";
                  #Reference.New = if #Reference.Ext = "Closure" then "Ref" else "" ]
   | use [ #Reference.Found = "Ref"; #Reference.New = if #Reference.Ext = "Closure" then "Def" else "" ]
```

Figure 2: Search Attributes.

```
Variable → Vdep Search Reference [
   - point is in, before, or after
   #Reference.Ext = #Search.Ext
   #Variable.Code = {
      NewInput ← $\#Search.Meet_{s \in \#Search.Prev(n)}$ Output[s]
      If NewInput ≠ Input[n] Then
         Input[n] ← NewInput
         FOUND ← {(v point s) ∋ (v point s)∈Input[n] and v∈#Reference.Found[n]}
         KILL ← {(v point s) ∋ (v point s)∈Input[n] and v = Def(n)}
         If FOUND ≠ $\phi$ Then
            For each (v point s) ∈ FOUND Do DD ← DD ∪ {(v point s, n)} Endfor
            Poutput[n] ← Input[n] - KILL;
            If #Search.Ext = "Closure" Then
               Soutput[n] ← Soutput[n] ∪ {(v #Search.Point n) ∋ v∈#Reference.New(n)}
            Endif
         Else Poutput[n] ← Input[n] - KILL Endif
         Output[n] ← Soutput[n] ∪ Poutput[n]
         If Output[n] ≠ $\phi$ Then
            For each s ∈ #Search.Next(n) Do Worklist → s $+_{\#Search.Order}$ Worklist Endfor
         Endif
      Endif } ]
```

Figure 3: Slices for Variable Dependences.

```
Statement → Sdep Search [
   #Statement.Code = {
      NewInput ← $\#Search.Meet_{s \in \#Search.Prev(n)}$ Output[s]
      If NewInput ≠ Input[n] Then
         Input[n] ← Poutput[n] ← NewInput
         If Input[n] ≠ $\phi$ Then
            DD ← DD ∪ {(s,n) ∋ s ∈ Input[n]};
            If #Search.Ext = "Closure" Then Soutput[n] ← Soutput[n] ∪ {n} Endif
         Endif
         Output[n] ← Soutput[n] ∪ Poutput[n];
         If Output[n] ≠ $\phi$ Then
            For each s ∈ #Search.Next(n) Do Worklist ← s $+_{\#Search.Order}$ Worklist Endfor
         Endif
      Endif } ]
```

Figure 4: Slices for Statement Dependences.

```
DDSpecify→ Variable | Statement [
        #DDSpecify.Code = {
            Begin
                 DD ← $\phi$;
                 WhileWorklist $\neq \phi$ Do
                        get $n$ from the head of Worklist;
                        #Variable.Code; | #Statement.Code
                 Endwhile
                 return(DD)
            End } ]
```

Figure 5: Slice Computation Loop.

```
Algorithm ComputeDataSlice ( ProgramPoints )
Input: Program flow graph;
Output: SLICE;
Declare: Worklist: ordered list of statement nodes;
         Soutput[n], Poutput[n], Output[n], Input[n], NewInput : set of variables
Begin
     SLICE = $\phi$
     For each statement n in the program flow graph Do
              Soutput[n] = Poutput[n] = Output[n] = Input[n] = $\phi$;
     Endfor
     For each "v before s" $\in$ ProgramPoints Do
              For each node n $\in$ Pred(s) Do
                       Soutput[n] = Soutput[n] $\cup$ {v}; Output[n] = Output[n] $\cup$ {v};
              Endfor
              Worklist ← s $+_{depth-first}$ Worklist
     Endfor
     For each "v after s" $\in$ ProgramPoints Do
              Soutput[s] = Soutput[s] $\cup$ {v}; Output[s] = Output[s] $\cup$ {v};
              For each node n $\in$ Succ(s) Do
                       Worklist ← n $+_{depth-first}$ Worklist
              Endfor
     Endfor
     While Worklist $\neq \phi$ Do
           get $n$ from the head of Worklist
           NewInput ← $\cup_{s \in Pred(n)}$ Output[s]
           If NewInput $\neq$ Input[n] Then
              Input[n] ← NewInput
              If Input[n] $\cap$ Ref[n] $\neq \phi$ Then
                 SLICE ← SLICE $\cup$ {n}
                 Poutput[n] ← Input[n] - Def(n)
                 Output[n] ← Soutput[n] $\cup$ Poutput[n]
              Else Output[n] ← Soutput[n] $\cup$ Input[n] Endif
              If Output[n] $\neq \phi$ Then
                 For each s $\in$ Succ(n) Do Worklist ← s $+_{depth-first}$ Worklist Endfor
              Endif
           Endif
     Endwhile
     return(SLICE)
EndComputeDataSlice
```

Figure 6: Algorithm Constructed from the specification **Construct DataSlice: forward immediate any use**;

Next we illustrate the construction and use of a forward partial analysis algorithm. Consider the specification "**Construct DD : Vdep forward immediate any use**". The partial analysis algorithm constructed from this specification, when given a set of variables at a program point, finds the future uses of those variable names. This algorithm only includes the code which deals with data dependencies and the extent of the search is limited to immediate data dependencies, the **Meet** operator is *union*, the **Next** set is the *Succ* set, the **Found** set is the *Ref* set, and the nodes in the *worklist* are ordered in depth-first order. The complete algorithm is given in Fig. 6.

Node	Worklist	Demand driven data flow
-	{S2,S3}	ϕ
S2	{S3,S4}	{(Y after S1, S2)}
S3	{S4}	{(Y after S1, S2),(X after S1, S3)}
S4	{S5}	{(Y after S1, S2),(X after S1, S3)}
S5	{S1,S6}	{(Y after S1, S2),(X after S1, S3)}
S1	{S6}	{(Y after S1, S2),(X after S1, S3), (Y after S1, S1)}
S6	ϕ	{(Y after S1, S2),(X after S1, S3), (Y after S1, S1),(Y after S1, S6)}

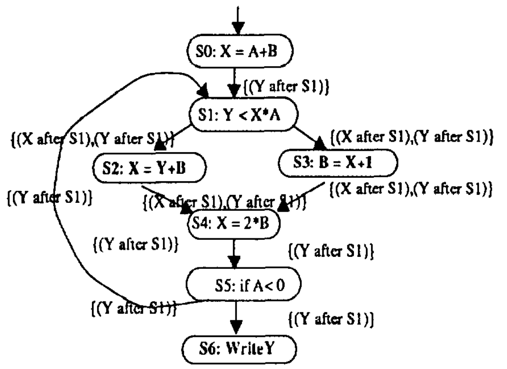

Figure 7. Computing forward variable dependencies: `Compute DD :- ( X after S1, Y after S1 ).`

The application of the partial analysis algorithm constructed for a sample program flow graph is illustrated in Fig. 7. The demand driven data flow is computed for variables X and Y after statement S1. The set Soutput[S1] is initialized to {(X after S1),(Y after S1)}. The worklist is initialized to the successor set of S1, that is, {S2,S3}. The search for uses of X and Y continues along the paths from S2 and S3. Since S2 uses Y and S3 uses X they are both included in the data flow set. As the search continues further the statement nodes for S4, S5, S1, and S6 are examined. This leads to the detection of uses of Y by S6 and S1. In Fig. 7 the worklist and the data flow set after the examination of each additional node is shown. Also the Input and Output sets of all the nodes are shown when the algorithm terminates. These sets at most contain both (X after S1) and (Y after S1) because the information was computed using these criteria. This illustrates how the computation of a partial data flow set is the computation of only the data flow information requested by the user.

4 Applications

Demand driven data flow information has been used to assist in the debugging and testing of programs. In the following sections, we give examples for debugging, testing and test case generation.

4.1 Test Case Generation and Regression Testing

In order to reduce the number of test cases generated when retesting a program after changes using data flow testing, statically determinable properties of a program, namely postdominance and dominance, can be used to guide the test case generation process [6, 7]. Although these properties can be efficiently determined exhaustively, program changes would require repeated computations of the properties. Our partial analysis can be used to avoid these computations after program changes. In particular, postdominance is a property that is used to group def-use pairs in an attempt to find, if possible, a test case that satisfies the entire group of def-use pairs. Since we need only the postdominators of a group of def-use pairs, and not all of the def-use pairs in the program, we use a partial postdominator algorithm to produce postdominators for the selected group of def-use pairs. Thus, after the def-use pairs have been identified as requiring test cases, partial analysis is performed to determine all the statements that postdominate def-use pairs in each group. Postdominator information can be obtained using statement dependencies. A statement S_1 is said to postdominate another statement S_2 if all paths from S_2 to the end of the program pass through S_1. The final node in a flow graph postdominates all nodes in the flow graph. The following statement dependency specification computes all nodes that are postdominated by statement S, and the postdominator relationships among those nodes. The search that is carried out is backward and the problem is an all paths problem.

```
Construct PDOMS: Sdep backward closure all
Compute PDOMS :- ( S )
```

As pairs are satisfied, partial analysis continues to reduce the size of the program considered. During the generation of a test case, for a group of def-use pairs that can be potentially tested by a single test case, we can make use of dominator information to guide the dynamic search that is likely to result in earlier termination. In other words, the effort expended in identifying the test case can be reduced through use of dominator information. As before, only partial dominator information is required. The forward statement dependencies specified below can be used to compute all statements that are dominated by statement S.

```
Construct DOMS: Sdep forward closure all
Compute DOMS :- ( S )
```

We have also demonstrated the utility of partial data flow algorithms in regression testing to find definition-use pairs that must be tested after a program change [8]. After changes are made to a previously tested program, regression testing attempts to retest only the changed portion of the code. We developed a data flow regression testing technique based on partial analysis. An important benefit of this approach is that, unlike previous techniques, no data flow history is needed nor is the recomputation of data flow for the entire program needed to detect changed definition-use pairs or values. The program changes drive the recomputation of the required data flow through partial analysis. The technique examines the various ways that a program can be modified, including inserting and deleting uses and definitions, changing operators in assignments and conditionals, and inserting and deleting edges. The technique employs two partial analysis algorithms, a forward and backward walk from the point of change help to identify the definitions and uses that are affected by a program edit. The backward walk identifies the definitions of a set of variables that reach the changed point. The specification for this demand driven data flow is:

```
BackwardWalk: Vdep backward immediate any def .
```

The forward walk identifies the uses of values that are directly or indirectly affected by a change in either a value or a predicate. The data dependencies required are computed by the following specification.

```
ForwardWalk: Vdep forward closure any use.
```

The data flow computed by the partial analysis algorithm generated by this specification is then integrated with control dependencies, as done in the slicing algorithm, to determine the affected pairs. Based on the type of change in the program, these two walks are used to find the def-use pairs that must be tested after a change.

4.2 Debugging

Various types of demand driven data flow information are particularly suitable for bug localization, including slicing [25]. For example, if the user knows that the value of a variable at a given point is incorrect, then the user can identify the statements that are potential sources of the bug by using *backward* data dependencies to identify the statements that directly or indirectly contributed to the computation of the incorrect value. The user may observe that the values of several variables are incorrect. These errors may be caused by a single bug or multiple bugs. Forward and backward dependencies can be useful in distinguishing between the two types of situations. One possible approach is to compute backward closure dependencies for each incorrect value. The intersection of all the backward sets will identify the statements that could have caused all of the errors. If there is no such statement then there are likely to be multiple bugs. Another approach would be to first identify the statements that potentially caused one of the variables to assume an incorrect value using a backward propagation. Next, using forward information, we can determine if the values of other variables known to be incorrect were directly or indirectly influenced by the same statements. If this is the case then we can conclude that the same bug may have caused several variables to have incorrect values. From the above description it should be clear that during debugging the user may compute several different types of demand driven data flow to isolate a bug. Both forward and backward dependencies are useful in bug localization and so are intersections and unions of different sets. Some examples of useful data flow sets are given in Fig. 8.

Type	Any Path		All Paths	
	Immediate	Closure	Immediate	Closure
Forward	I	II	V	VI
Backward	III	IV	VII	VIII

S: X = ..Y..;

I: Uses of X reachable by X's definition in S along any path:
Construct DD : Vdep forward immediate any use
Compute DD :- (X after S)

II: All statements that may be affected by X's definition in S:
Construct DD : Vdep forward closure any use
Compute DD :- (X after S)

III: Definitions of Y that reach Y's use in S along any path:
Construct DD : Vdep backward immediate any def
Compute DD :- (Y before S)

IV: All statements that might affect the value of Y in S:
Construct DD : Vdep backward closure any def
Compute DD :- (Y before S)

V: Those Uses of X reachable by only X's definition in S along all paths:
Construct DD : Vdep forward immediate all use
Compute DD :- (X after S)

VI: Those statements that are affected by only X's definition in S:
Construct DD : Vdep forward closure all use
Compute DD :- (X after S)

VII: Those definitions of Y that, if executed, reach only Y's use in S along all paths:
Construct DD : Vdep backward immediate all def
Compute DD :- (Y before S)

VIII: Those statements that, if executed, affect only the value of Y in S:
Construct DD : Vdep backward closure all def
Compute DD :- (Y before S)

Figure 8: Examples of Demand Driven Data Flow Useful for Debugging and Testing Programs.

If the bug in the program is in the form of an incorrect assignment of a value to a variable, then data dependencies are sufficient to identify the bug. However, the cause of a bug maybe an incorrectly specified predicate in a conditional statement. In such situations control dependence information may be needed. Control dependencies can be integrated with the information derived from the partial analysis of data dependencies. As an example, consider the Weiser slice which consists of a subprogram whose execution is sufficient to enable the computation of the value of a variable at a given program point. Weiser's slice is computed by performing a backward closure over data and control dependencies starting from the point of interest. The reaching definitions that capture the relevant data dependencies are computed in a demand-driven fashion as shown below.

```
Construct DataSlice: Vdep backward closure any def
WeiserSlice = $\phi$
Input = {v after/before s, ....}
repeat
        Compute DataSlice :- (Input)
        WeiserSlice = WeiserSlice ∪ DataSlice
        ControlSlice = {S: ∃ S' ∈ WeiserSlice, where S'
                is control dependent upon S }
        WeiserSlice = WeiserSlice ∪ ControlSlice
        Input = ControlSlice
until Input = $\phi$
```

The Weiser's slice can also be computed by a backward walk along control and data dependence edges on a program dependence graph. However, this approach requires exhaustive computation of data flow information since the program dependence graph can only be constructed after all data dependencies and control dependencies have been computed. Also, as was indicated, slices from program points other than where a variable is used are not available (e.g., in Fig. 7, slice at S4 on Y).

5 Related Work

The framework presented in this paper differs from the usual data flow framework [12] in a number of important ways. First of all, the number and complexity of parameters are increased due to the selection of the starting point and variables that is an inherent property of demand driven data flow. Another difference is that we also have a language for expressing our characteristics that is of a higher level than the parameters needed in the algorithms. The parameters that are actually needed in the algorithm are derived from these characteristics.

One type of demand driven data flow that has been developed and used extensively is the static slice [14]. Slicing as defined by Weiser is a data flow analysis technique that computes the set of statements contributing to the dependency information desired for a given slicing criterion. The value of slicing is that it filters out statements that are not needed to address the questions being asked about the behavior of the program.

Related to partial analysis is incremental analysis in terms of overall goals. However, the focus of incremental data flow analysis is to maintain a global data flow solution by incrementally updating the solution in response to small changes in the program [1, 18, 21, 22, 26]. It may be too expensive to fully reanalyze a program from scratch each time a small change is made to the program. Instead, an incremental data flow algorithm takes the global solution to an initial instance I of a data flow problem and a small change from instance I to another instance I' and computes the updated solution for the changed instance I'.

Olender and Osterweil have developed a tool that automatically performs static interprocedural sequencing analysis from programmable constraint specifications [15, 16]. Sequencing analysis enables the detection of data anomalies in a program. In order to efficiently carry out sequencing analysis for various objects, a program slice is computed for each object. The slice for a given object is represented as a reduced control flow graph which can be expected to be significantly smaller than the entire program flow graph. Thus, sequencing analysis of objects using slices is more time efficient than an implementation that relies on the original control flow graph. However, it should be noted that this approach requires large amount of storage to store precomputed slices. In the proposed research we are attempting to improve *both* time and space efficiency of data flow analysis. Furthermore, the above approach requires the precomputation of slices for all objects.

Work related to the construction of data flow algorithms has been addressed in the Sharlit tool that was developed to help compiler writers develop optimizers and data flow analyzers [23]. Abstractions are presented that enable compiler writers to program in a modular fashion. Our technique is oriented toward demand driven data flow computation (although exhaustive data flow can be computed), employs specifications to automatically produce algorithms and is oriented to software engineering tools as well as compilers. SPARE is another tool that facilitates the development of program analysis algorithms [24]. This tool supports a high-level specification language through which analysis algorithms are expressed. The denotational nature of the specifications enables automatic implementation as well as verification of the algorithms. Although the tool is useful for rapid prototyping, the efficiency of the automatic implementations may not be acceptable in a production environment.

6 Implementation

We have implemented our technique and have produced various types of partial analysis algorithms. In preliminary experiments performed, we evaluated the merit of using partial analysis algorithms to compute demand driven data flow information. We made simple modifications in programs and computed the change in data flow using partial analysis algorithms as compared to exhaustively recomputing all data flow sets. The results based upon a sample of our runs indicate a large savings in the size of the data flow sets in addition to a savings in execution time. In general, the data flow sets, which can grow to very large sets, were reduced. The size of the data flow sets when using partial data flow ranged from 1 element to 8 elements as compared with sets ranging from 49 to 73 elements for exhaustive data flow analysis. The execution time savings for the programs was about 20%. The programs used were fairly small programs and we expect better performance, at least in space, when larger programs are used.

References

[1] M. Burke, "An interval analysis approach toward exhaustive and incremental interprocedural data flow analysis," *Technical Report RC 12702*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, July 1987.

[2] D. Callahan and J. Subhlok, "Static analysis of low-level synchronization," *Proceeding of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, SIGPLAN Notices, Vol. 24, No. 1, pages 100-111, January 1989.

[3] E. Duesterwald, R. Gupta and M.L. Soffa, "Rigorous data flow testing through output influences," *Proc. 2nd Irvine Software Symposium*, pages 131-145, Irvine, CA, March 1992.

[4] E. Duesterwald and M.L. Soffa, "Static concurrency analysis in the presence of procedures using a data-flow framework," *Proc. ACM Symposium on Testing, Analysis, and Verification*, Victoria, British Columbia, pages 36-48, October 1991.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pages 319-349, July 1987.

[6] R. Gupta and M.L. Soffa, "Employing static information in the generation of test cases," *Journal of Software Testing, Verification and Reliability*, Vol. 3, No. 1, pages 29-48, December 1993.

[7] R. Gupta, "Generalized dominators and postdominators," *Proc. 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246-257, Albuquerque, New Mexico, January 1992.

[8] R. Gupta, M.J. Harrold, and M.L. Soffa, "An approach to regression testing using slicing," *Proc. Conference on Software Maintenance*, Orlando, Florida, pages 299-308, November 1992.

[9] M.J. Harrold and M.L. Soffa, "Interprocedural data flow testing," *Proc. ACM Symposium on Software Testing, Analysis and Verification*, pages 158-167, Key West, Florida, December 1989.

[10] S. Horwitz, J. Prins and T. Reps, "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, pages 345-387, July 1989.

[11] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 1, pages 26-60, January 1990.

[12] J.B. Kam and J.D. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica*, Vol. 7, pages 305-317, 1977.

[13] W. Landi and B. Ryder, "A safe approximation algorithm for interprocedural pointer aliasing," *Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation*, pages 235-248, June 1992.

[14] J.R. Lyle and M. Weiser, "Automatic program bug location by program slicing," *Proc. Second IEEE Symposium on Computers and Applications*, pages 877-883, June 1987.

[15] K.M. Olender and L.J. Osterweil, "Interprocedural static analysis of sequencing constraints," *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 1, pages 21-52, January 1992.

[16] K.M. Olender and L.J. Osterweil, "Cecil: a sequencing constraint language for automatic static analysis generation," *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, March 1990.

[17] D. Padua and M.J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, Vol. 22, No. 12, pages 1184-1201, December 1986.

[18] L.L. Pollock and M.L. Soffa, "An incremental version of iterative data flow analysis," *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, pages 1537-1549, December 1989.

[19] S. Rapps and E. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, pages 367-375, April 1985.

[20] T. Reps, T. Teitelbaum and A. Demers, "Incremental context-dependent analysis for language-based editors," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, pages 449-477, July 1983.

[21] B. Rosen, "Linear cost is sometimes quadratic," *Proc. Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 117-124, June 1981.

[22] B.G. Ryder and M. C. Paull, "Incremental data flow analysis algorithms," *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 1, pages 1-50, Jan. 1988.

[23] S.W.K. Tjiang and J.L. Hennessy, "Sharlit - a tool for building optimizers," *Proc. ACM Sigplan Conf. on Programming Language Design and Implementation*, pages 82-93, 1992.

[24] A. Venkatesh and C.N. Fischer, "SPARE: a development environment for program analysis algorithms," *IEEE Transactions on Software Engineering*, Vol. 18, No. 4, April 1992.

[25] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pages 352-357, July 1984.

[26] F. Zadeck, "Incremental data flow analysis in a structured program editor," *Proc. ACM SIGPLAN 1984 Symposium on Compiler Construction*, June 1984.