INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

UTILIZING THE NOTION OF REMOVABLE BLOCKS TO ENHANCE PROGRAM SLICING ALGORITHMS

Bhaskar Airody Karanth

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

June 2001

0-612-64084-1

ABSTRACT

Utilizing the notion of removable blocks to enhance program slicing algorithms

Bhaskar Airody Karanth

Program slicing is a program decomposition technique that transforms a large program into a smaller one that contains only statements relevant to the computation of a selected function. Applications of program slicing can be found in software testing, debugging and maintenance where program slicing essentially reduces the amount of data that has to be analyzed in order to comprehend a program or parts of its functionality. In this thesis, two program slicing algorithms based on the notion of removable blocks are presented and they are (1) a general static program slicing algorithm and (2) criterion based hybrid program slicing algorithm. The thesis introduced new syntax tree representation using removable blocks and theorized a new navigation technique for the same. The new static slicing algorithm combined with the enhanced dynamic slicing algorithm is used to derive the criterion based hybrid slicing algorithm. The hybrid program slicing algorithm allows the user to define the range of accuracy between static and dynamic program slicing. These algorithms compute slices that are executable for structured and object-oriented programs. It uses the executable property of new static slicing to reduce the input source code to dynamic slicing to save the time and space. The introduced program slicing approaches are part of Montreal Object-Oriented Slicing Environment (MOOSE). MOOSE utilizes the information derived from the program slicing algorithms to enhance the functionality and usability of the framework. The preliminary tests with the basic hybrid program slicing algorithm indicate that hybrid program slicing can reduce the algorithmic time and space for the slice computation as compared to the dynamic program slicing.

ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Juergen Rilling, thesis supervisor, for all his patience and guidance throughout my research work. His supervision and support proved instrumental in helping me to complete what at times seemed to be an insurmountable task.

I gratefully acknowledge Hatch & Associates, Montreal, for their encouragement and understanding during my thesis work. Many thanks to P.Girard, Associate, without his consistent support, this thesis would have been an uphill task.

I wish to thank my wife and children for encouraging me to pursue the second graduate degree while allowing themselves a lot of discomfort during this period. I would like to thank all the fellow students who have contributed to my knowledge during the last four years of study at Concordia University.

Table of Contents

List of figures

1. Introduction

The comprehension of source code plays a prominent role during software maintenance and evolution. Poor design, unstructured programming methods, and crisis-driven maintenance can contribute to poor quality code, which in turn affects program comprehension. The goal of program comprehension is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner. There are varieties of support mechanisms for aiding program comprehension, which can be grouped into three categories: unaided browsing, leveraging corporate knowledge, experience, and computer-aided techniques like reverse engineering. In this thesis, focus is on reverse engineering as it can be applied effectively in program comprehension [3, 18, 45].

One approach to improve the comprehension of programs is to reduce the amount of data that has to be observed and inspected. In this research, program slicing is utilized to enhance the comprehension of software systems. The notion of static program slicing originated in the seminal paper by Weiser [56,57]. Weiser defined a slice *S* as a reduced, executable program obtained from a program *P* by removing statements such that *S* replicates parts of the behavior of *P*. Weiser's approach is based on program dependencies; slices are consecutive sets of indirectly relevant statements. A static program slice consists of those parts of a program *P* that potentially could affect the value of a variable *v* at a point of interest. The static algorithm uses only statically available information for the slice computation; hence, this type of program slicing is referred to as a static slice. Different extensions of the original static slicing approach have been proposed, e.g., [11]. Korel introduced a major extension of program slicing, with Laski

[23], called dynamic slicing. The dynamic slicing approach not only utilizes static source code information, but also dynamic information from program executions on some program input. The dynamic slice preserves the program behavior for a *specific* input, in contrast to the static approach, which preserves the program behavior for the set of all inputs for which the program terminates. By considering only a particular program execution rather than all possible executions, dynamic algorithms may compute slices that are significantly smaller than the slices computed by the static slicing algorithms. Different types of dynamic program slices have been proposed, e.g., [2,4,13,21,23,28]. The reason for this diversity of slicing types and methods is the fact that different applications require different properties of slices. The notion of dynamic slicing has also been extended for distributed programs [7,10,26]. Program slicing is not only used in software debugging but also in software maintenance and software testing [3,14,15,17,22,27,33,38,44,54,58]. Slicing has been shown useful in program debugging [8,23,48,56], testing, and program comprehension and software maintenance [1,5,12,20,24].

Hybrid program slicing algorithms were introduced to take advantage of both static and dynamic slicing properties. These algorithms use static information to lower the run time overheads and dynamic information is used for more accurate handling of dependencies [47]. Gupta and Souffa proposed in [15] to use both static and dynamic information for the computation of program slices for structured programs. Schoenig and Ducass'e in [47] proposed a hybrid backward slicing algorithm for Prolog, which computes an executable slice.

1.1 Motivation and objective

In this thesis, the objective is to investigate enhanced program slicing and its usability in program comprehension. The underlying principle of removable blocks is used in the theorized program slicing algorithms. Korel and Laski in [23] introduced dynamic program slicing with removable blocks but did not extend the same notion for static program slicing. This has prompted the current research to hypothesize a new algorithm for static program slicing based on removable blocks.

The process of program comprehension can become aggravating in software maintenance and debugging because programmers usually debug someone else's programs and often, they only poorly or partially understand these programs. A new general static program slicing algorithm is presented in the current research that extends the algorithm for unstructured programs based on Korel's algorithms [30]. This algorithm can be adapted for all programming language constructs as they can be found in major procedural programming languages, e.g.; procedures, functions, recursion, nested procedure calls, exit, abort, exception handling, local and global variables. In addition, a formal proof is presented to show that the new general static program slicing algorithm computes correct slices.

A hypothesis is proposed in this research, that developing novel static and hybrid program slicing related concepts would help the comprehension by reducing the time and resources required for the generation of precise slices for a range of values. As part of a program slice computation, different types of information are determined and usually discarded after the slice computation. In this research, new hybrid program slicing related features are proposed to exploit this information for the purpose of program comprehension and comprehension of program executions, e.g., executable hybrid program slices, partial static and hybrid program slicing. Two hybrid program slicing concepts are suggested in this research and they are known as "Basic Hybrid Program slicing Algorithm (BHPSA)" and "Criterion Based Hybrid Program Slicing Algorithm (CBHPSA). For BHPSA, both static and dynamic program slicing algorithms were used sequentially. Similar to static program slicing, a formal proof is presented to show that the criterion based hybrid program slicing algorithm computes correct slices.

The BHPSA has been implemented within MOOSE (Montreal Object-Oriented Slicing Environment). MOOSE was developed as an open comprehension framework to guide programmers during the challenging task of understanding large traditional and object-oriented programs and their executions. Certain number of tests were carried out to understand the behavior of such hybrid program slicing algorithms.

1.2 Scope of the dissertation

The presented thesis consists of six sections including this section.

In section two, an overview and a survey of related literature and existing approaches of static, dynamic and hybrid slicing algorithms is provided, as well as a general comparison of these algorithms.

In the third section, a general static program slicing algorithm based on removable blocks is presented. A formal proof of the algorithm is presented and it is based on Korel's dynamic algorithm [30].

In the fourth section, two novel hybrid programs slicing related concepts on the source code level is presented. Also presented is an algorithm on criterion based hybrid slicing that is an extension to Korel's dynamic program slicing algorithm [30].

In the fifth section, an overview of MOOSE environment is presented. A brief note on, preliminary tests that are conducted using MOOSE environment are shown. In addition, in this section, properties on which further experiments are to be carried out are outlined for future work.

In the last section, conclusions of the present work and propose future directions related to this research.

In appendix A, the formal proof of general dynamic slicing from Korel [30] is reproduced to help the readers to understand the concept removable blocks applied to dynamic slicing.

2. Background

Program comprehension is a crucial part of system development and software maintenance. It is expected that a major share of systems development effort go into modifying and extending pre-existing systems, about which programmer usually know little. Change to a system may be necessitated for adaptive, perfective, corrective or preventive reasons. Understanding the system, incorporating the change, and testing the system to ensure that the change has no united effect on the system are the three facets of software maintenance [13,35,53]. For all of these maintenance activities, software comprehension plays a pivotal role. A commonly used technique to enhance the comprehensibility of software systems is through reverse engineering. This technique is used to analyze a subject system with the goal to: (a) identify the system's components and their inter-relationships (b) to create representations of a system in another form at a higher level of abstraction and (c) to understand the program execution and the sequence in which it occurred. Numerous theories have been formulated and empirical studies are conducted to explain and document the problem-solving behavior of software engineers engaged in program comprehension. Cognitive models have been introduced to describe the comprehension processes and knowledge structures used to form a mental representation of the program under examination [39].

Typically, a program performs a large set of functions/outputs. Rather than trying to comprehend all of a program's functionality, programmers will focus on selected functions (outputs) with the goal to identify which parts of the program are relevant for that particular function. Program slicing provides support during program comprehension, by capturing the computation of a chosen set of variables/functions at some point (static slicing) in the original program or at a particular program execution position (dynamic slicing). This will lead to a smaller, simplified version of the original version of the program without changing the local semantics of the extracted slice.

2.1 Basic program slicing terminology

Program slicing terminology is based on program dependence theory and it reuses its terminology. Most of the slicing algorithms are represented by a directed graph, which captures the notion of data dependence and control dependence in programs. The program structure is represented by a flow graph $G = (N, A, s, e)$ where (1) N is a set of nodes, (2) A, a set of arcs, is a binary relation on N and (3) s and e are, respectively, unique entry and exit nodes. A node corresponds to an assignment statement, an input or output statement or the predicate of a conditional or a loop statement, in which case it is called a *test* node. A *path* from the entry node s to some node k, $k \in N$, is a sequence $<n_1, n_2, n_q>$ of nodes such that $n_1 = s$, $n_q = k$ and $(n_i, n_{i+1}) \in A$, for all n_i, $1 \leq i < q$. A path that has actually been executed for some input will be referred to as an *execution trace.* A path is regarded feasible only if there exists some input data, which causes the path to be traversed during a particular program execution. A program trajectory has been defined as a feasible path that has actually been executed for some specific input. Notationally, an *execution trace* is an abstract list (sequence) whose elements are accessed by position in it, e.g., for trace T_x in Figure 2, $T_x(4)=4$, $T_x(5)=8$. Node Y at position p in T_x (e.g., $T_x(p)=Y$) will be written as Y^p and referred to as an *action.* Y^p is a *test* action if Y is a test node v^q denotes *variable v at position q*, i.e., variable (object) v before execution of node $T_x(q)$.

For example, T_x = <1,2,3,4,9,10,11,12> is the execution trace when the program in Figure 1 is executed on the input x: *MSRP* = 25000; this execution trace is presented in Figure 2.

```
1. normal_profit       = 100;
2. bonus_profit        = 1500;
3. cin >>MSRP;
4. if (MSRP > 30000)
   {
5.        bonus_profit     = MSRP + bonus_profit ;
6.        normal_profit   = MSRP + normal_profit ;
7.        Showroom_price  = MSRP + bonus_profit + normal_profit;
   }
8   else
   {
9.        Showroom_price = MSRP + normal_profit ;
   }
10.  cout<< normal_profit;
11.cout <<bonus_profit;
12.cout<< Showroom_price;
```

Note: Compiler specific declarations are excluded for clarity

Figure 1: Sample program

1^{1}	`normal_profit  = 100`
2^{2}	`bonus_profit   = 1500;`
3^{3}	`cin >>MSRP;`
4^{4}	`if (MSRP > 30000)`
8^{5}	`else`
9^{6}	`Showroom_price = MSRP +normal_profit ;`
10^{7}	`cout<< normal_profit;`
11^{8}	`cout <<bonus_profit;`
12^{9}	`cout<< Showroom_price;`

Figure 2: An execution trace of the sample program on input MSRP = 25000

A *use* of variable v is an action in which this specific variable is referenced. A *definition* of variable v is an action, which assigns a value to that variable. The following assumptions are made: $U(Y^p)$ is a set of variables whose values are used in action Y^p and $D(Y^p)$ is a set of variables whose values are defined in Y^p. Sets $U(Y^p)$ and $D(Y^p)$ are determined during program execution, especially for array and pointer variables because it is possible to identify the specific array elements that are used or modified by the action during program execution.

Static data dependence captures the situation in which one node assigns a value to an item of data and the other action uses that value. Data dependence is based on the concepts of a variable definition and use. Thus a node j is data dependent on node i if there exists a variable v such that: (1) v is defined in node i, (2) v is used in node j and (3) there exists a path from i to j without an intervening definition of variable v. In the sample program of Figure 1 there exists data dependence between node 5 (using the variable *MSRP)* and node 3 (defining the variable *MSRP).*

Static control dependence is based on the concepts of post-dominance. Informally this can be thought of as one program statement determining in some way, whether or not another statement will be executed. The control dependence is defined as [11]: Let *Y* and *Z* be two nodes and *(Y, X)* be a branch of Y. Node Z *postdominates* node *Y* iff *Z* is on every path from *Y* to the exit node *e*. Node *Z* post-dominates branch *(Y, X)* iff *Z* is on every path from *Y* to the program exit node *e* through branch *(Y, X)*. *Z* is *control dependent* on *Y* iff *Z* post-dominates one of the branches of *Y* and *Z* and does not post-dominate *Y*. The concept of post dominance means that all execution paths in a control flow graph from a specific node i to the program end must pass through another node j before they reach the program end [19]. For example, in the sample program of Figure 1, there exists a control dependence between the node 5 and node 4, where node 5 is control dependent on node 4.

2.1.1 Backward program slicing

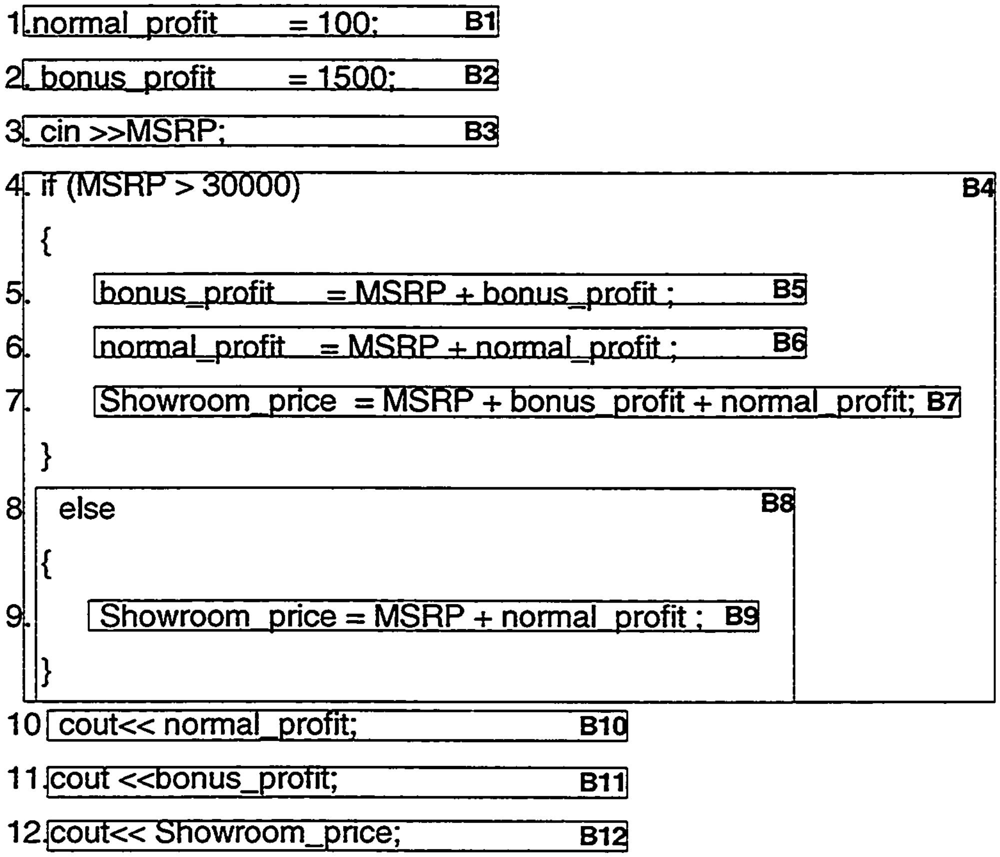
In backward slicing, slices are computed by gathering statements and control predicates through backward traversal of the program dependencies, starting at the node defined by the slicing criterion [54]. The slices are mostly obtained by traversing the edges of graph towards the root node.

2.1.2 Forward program slicing

A forward program slice contains all statements and control predicates dependent on the slicing criterion. A statement is 'dependent' on the slicing criterion: 1) if the value computed at that statement depend on the values computed at the slicing criterion or 2) if the values computed at the slicing criterion determine the fact if the statement under consideration is executed or not.

2.1.3 Removable Blocks

Korel introduced the notion of removable blocks in [28] and described it as the part of program text (code) that can be removed during slice computation. A block is described as the smallest component of the program text that can be removed (e.g. assignment statement, input and output statements, etc.). Test nodes (predicates of conditional statements) are not removable individually, and, therefore, they are considered part of a complex block where they can be removed if none other block in the complex block is said to be not removable. Intuitively, a block may be removed from a program if its removal does not "disrupt" the flow execution on some input x. Each block B has a regular entry to B and a regular exit from B referred to as *r-entry* and *r-exit,* respectively. In unstructured programs, because of jump statements, execution may enter a block directly without going through its *r-entry*; in this case, one can say execution enters the block through a *jump entry*. Similarly, execution can exit a block without going through its *r-exit*; in this case, the execution leaves a block through a *jump exit*. Let B_1; B_2; B_3; be a sequence of three blocks in a program. Block B_2 may be removed, if during execution of the program on some input x, the execution exits from block B_1 through its *r-exit*, enters block B_2 through its *r-entry*, leaves B_2 through its *r-exit*, and enters block B_3 through its *r-entry*. If block B_2 is removed and the resulting program is executed on the same input x, the program execution will after leaving B_1 through *r-exit*, enter block B_3 directly through its *r-entry*. In this case, the flow of execution is not disrupted by the removal of block B_2. Figure 3 shows the sample program represented in Korel's removable block concept.



```
normal_profit      = 100;
bonus_profit       = 1500;
cin >>MSRP;
if (MSRP > 30000)
{
     bonus_profit     = MSRP + bonus_profit ;
     normal_profit    = MSRP + normal_profit ;
     Showroom_price  = MSRP + bonus_profit + normal_profit;
}
 else
{
     Showroom_price = MSRP + normal_profit ;
}
cout<< normal_profit;
cout <<bonus_profit;
cout<< Showroom_price;
```

Figure 3: Sample program(Figure 1) with removable blocks

2.2 Static program slicing

Based on the original definition of Weiser [57] a static program slice *S* consists of all statements in program *P* that may affect the value of variable *v* at some point *p*. The slice is defined for a slicing criterion *C=(x,V)*, where *x* is a statement in program *P* and *V* is a subset of variables in *P*. Given *C*, the slice consists of all statements in *P* that potentially affect variables in *V* at position *x*. Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies. The program dependence graph (PDG) was originally defined by Ottenstein and Ottenstein [40] and later refined by Horwitz et al. [19,41,42]. Data and control dependencies between nodes may form a program dependence graph that can be used for the computation of static slices by traversing backwards along the edges of the program dependence graph from a point of interest.

2.2.1 Program dependency graph(PDG)

Program dependence graph is formed by combining all data and control dependencies that exist in the program. The existing static program slicing algorithms use the notion of data and control dependencies to compute program slices. A static program slice can be easily constructed using the PDG by traversing backwards along the edges of a program dependence graph starting at a node *i*. Figure 4 shows the program dependency graph for the sample program shown in Figure 1. The nodes, which were visited during the traversal, constitute the desired slice [40]. Figure 5 shows the executable slice created by the static slicing algorithm for the sample program in Figure 1. For this example, one need to first traverse backward through all nodes with edges from the starting node 12 (variable *Showroom_price* for which the slice is computed). The edges which can be traversed backwards from node 12 through the PDG are the node 7 and node 9.

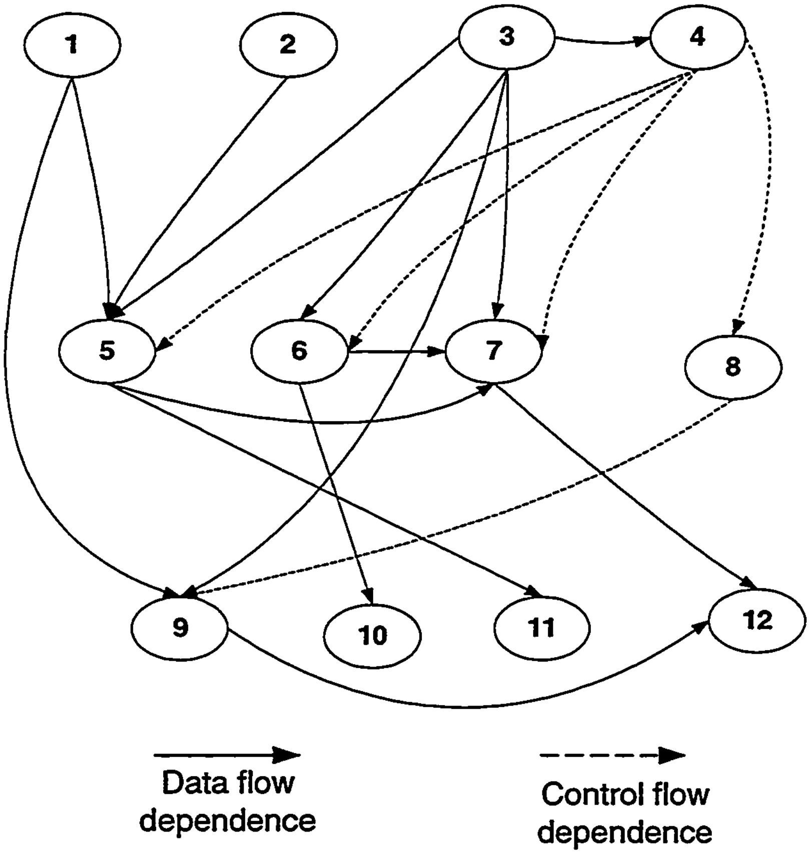


Figure 4: Program dependence graph for the sample program from Figure 1

In the next step all the edges which can be traversed backward from node 7 and node 9 are visited. The set of all nodes which could be traversed backwards through the PDG from starting node 12 are shown in Figure 4. This set of nodes {1,2,3,4,5,6,7,8,9,12}, represents the static program slice for variable *v* at node *i*, which includes all statements affecting the variable *Showroom_price* at node 12. For this example, the static program slice is only slightly smaller than the original sample program (see Figure 5). The only statements, which are not included in the static program slice, are the output statements at nodes 10 and 11.

```
1^1.    normal_profit = 100;
2^2.    bonus_profit  = 1500;
3^3.    cin >>MSRP;
4^4     if (MSRP > 30000)
        {
5^5.      bonus_profit = MSRP + bonus_profit ;
6^6.      normal_profit =  MSRP + normal_profit ;
7^7.      Showroom_price  = MSRP + bonus_profit + normal_profit;
        }
8^8.    else
        {
9^9.       Showroom_price = MSRP +normal_profit ;
        }
12^10.  cout <<Showroom_price;
```

Figure 5: Static slice for *Showroom_price* on input MSRP = 25000

2.2.2 Data Flow Based Slicing Techniques

The two algorithms based on Weiser's [57] original data-flow model for computing program slices is presented by Gupta et al. [14], which they describe "*backward*" and "*forward*" walks. These algorithms are based on the concept of "*backward*" and "*forward*" program slicing. They identify the *definitions* and *uses* that are affected by a program. Both algorithms use a control flow graph representation of the program in which each node represents a single statement. The algorithms compute data flow information to identify the affected def-use associations, but they do not require history of data flow information. Furthermore, the algorithms are slicing algorithms in that they examine only relevant parts of the control flow graph to compute the required data flow information. These algorithms are designed based on the approach taken by Weiser for computing slices [57]. This approach lets relevant program slices be computed without exhaustively computing the def-use information for the program. The algorithms assume that only scalars are being considered and the technique is easily extended to include arrays by adding a new condition for halting the search along paths. The algorithms from [14] are presented below explain the concept.

Algorithm *BackwardWalk*[14]

The backward walk algorithm identifies statements containing definitions of variables that reach a program point. The technique uses the definitions that reach the statement, along with the uses in the statement, to form def-use associations. It should be noted that if a variable being considered is undefined along a path, then the search will terminate once it reaches the start node of the control flow graph. Algorithm *BackwardWalk*, shown in Figure 6, identifies the statements containing definitions of a set *U* of variables that reach a program point *s*. *BackwardWalk* inputs the program point or statement *s* and a set *U* of program variables, and outputs *DefsOf*U, a set of statements or nodes in the control flow graph corresponding to the definitions of variables in *U* that reach *s*. *BackwardWalk* traverses the control flow graph in the backward direction from *s* until all variables of *U* are encountered along each path. The algorithm collects the statements containing the definitions in *DefsOfU* and returns the set.

To assist in the traversal process, *BackwardWalk* maintains two additional sets of variables, *In* and *Out*, for relevant nodes in the control flow graph. *Out*[*i*] contains the variables whose definitions were not encountered along some path from the point immediately following *i* to *s*; *In*[*i*]contains the variables whose definitions the algorithm has not encountered along some path from the point immediately preceding *i* to *s*. Since the algorithm walks backward in the control flow graph, it computes *Out*[*n*]as the union of the *In* sets of *n*'s successors. The algorithm uses another set of variables, *NewOut*, to store temporarily the newly computed *Out* set. During the backward traversal, the algorithm maintains a *Worklist*, consisting of those nodes that must be visited; *Worklist* indicates how far the traversal has progressed. *BackwardWalk maintains Worklist* as a priority queue based on a reverse depth first ordering of nodes in the control flow graph.

```
algorithm BackwardWalk(s,U)
input          s :program point/statement
               U :set of program variables
output  DefsOfU :set of statements/nodes in the control flow graph
declare  In[i], Out[i], NewOut:set of program variables
               Worklist :statements/nodes in the control flow graph, maintained as a priority queue
               n, $n_i$ :program point/statement
               Pred(i), Succ(i):returns the set of immediate predecessors(successors) of i
begin
        DefsOfU = Worklist = ∅
        forall n ε Pred(s) do Worklist = $n_{rdf}$+ Worklist
               In[s]=U;   Out[s]=∅
        forall ni ≠ s do In[$n_i$ ]=Out[$n_i$ ]=∅
        while Worklist ≠ ∅ do
               Get n from head of Worklist
               NewOut $=_{p\, \varepsilon\, Succ(n)} \cup$ In[p]
        if NewOut ≠ Out[n] then
               Out[n]=NewOut
        if n defines a variable u ε U then
               DefsOfU = DefsOfU ∪ {n}
               In[n]=Out[n]-{u}
        else In[n]=Out[n]
        if In[n] ≠ ∅ then
               forall x ε Preds(n) do Worklist = $x_{rdf}$+ Worklist
        return(DefsOfU)
end BackwardWalk
```

Figure 6: Algorithm *BackwardWalk* [14]

The algorithm also uses *n* and n_i to represent statements or nodes in the control flow graph, and functions *Pred(i)* and *Succ(i)* to compute the immediate predecessors and successors of node *i*, respectively.

Algorithm *BackwardWalk* begins by initializing all sets that it uses. After initialization, the only entries in *Worklist* are the predecessors of *s*. The main part of the algorithm is a **while** loop that repeatedly processes statements in *Worklist* until *Worklist* is empty. To process a statement n, *BackwardWalk* first computes *NewOut* for *n* as the union of the *In* sets of the successors of *n* in the control flow graph. If *NewOut* and *Out[n]* are the same, there has been no change from the last iteration of the **while** loop, and processing along the path containing *n* terminates; the comparison of *NewOut* and *Out[n]* causes each loop to be processed only one time. If *NewOut* and *Out[n]* differ, there is a change from the last iteration of the **while** loop. In this case, *BackwardWalk* assigns *NewOut* to *Out[n]*, and examines *n* for a definition of a variable in U. If the algorithm finds such a definition, it adds *n* to *DefsOf*U

Additionally, the algorithm removes *u* from *In[n]* since it no longer needs to search for a definition of *u* along this path. If *BackwardWalk* finds no definition of a variable in *U* in *n*, it assigns *Out[n]* to *In[n]* and adds all immediate predecessors of *n* to *Worklist*. Each statement *n* added to *Worklist* represents a point in the program along which the backward traversal must continue, since not all variables in *U* were defined along a path from a successor of *n* to point *s*. Thus, *BackwardWalk* only adds a node to *Worklist* if the *In* set of one of its successor is not empty. When *Worklist* is empty, the algorithm has encountered all definitions of all variables in *U* along all backward paths from *s* and the algorithm terminates.

Algorithm *ForwardWalk*[14]

The forward walk algorithm identifies uses of variables that are affected directly or indirectly by either a change in a value of a variable at a point in the program or a change in a predicate. The def-use associations returned by the algorithm are triples *(s, u, v)* indicating that the value of variable *v* at statement *s*, affected by the change, is used by statement u. A def-use association is directly affected if the triple represents a use of an altered definition. A def-use association is indirectly affected in one of the two ways: (1) the triple is in the transitive closure of the changed definition or (2) the triple is control dependent on a changed or affected predicate.

Algorithm *ForwardWalk*, presented in Figure 7, inputs a set of *Pairs* representing definitions whose uses are to be found, along with a Boolean, *Name*s, that indicates whether the walk starts with a set of variable names at a program point or a set of definitions. *Names* is true if the walk begins with a set of variable names *v* at a point p, represented by *(p, v)*. Otherwise, the walk begins with the pairs of affected definitions *(s,v)*. *ForwardWalk* outputs a set of def-use triples, *Triple*s. For each statement node *n*, *In* and *Out* sets contain the pairs representing definitions whose uses are to be found, since their values are affected by the edit. The set *In[n](Out[n])* contains the values just before (after) *n* whose uses are to be found. Each value is represented as a pair *(d,p)*indicating that the value of variable *p* at point *d* is of interest. If *ForwardWalk* encounters a statement *n* that uses the value *(d,p)* belonging to *In[n]*, it adds def-use triple *(d,n,p)*to the list of def-use pairs affected by a change in the value of variable *v* at statement *s*. The value of the variable defined by statement *n* is also indirectly affected.

```
algorithmForwardWalk(Pairs, Names)
input   Names :boolean is true if change is only a predicate change
        Pairs :sets of definitions, (s, v), where s is a program point/statement and v is a variable
output  Triples :set of (point/statement, statement, variable)
declare In[i], Out[i], Kill, NewIn :set of pairs, point/statement,variable)
            Worklist, Cd[i], PredCd, AffectedPreds :set of point/statement
            DefsOfV :set of (s, v)of definitions
            v :program variable
            k, n :statement/node
          Pred(i), Succ(i):returns the predecessors(successors) of i in the control flow graph
          Def (i): returns the variable defined by statement I
begin
        Triples = $\emptyset$
        forall (s,v) $\varepsilon$Pairs do
                forall n $\varepsilon$ Succ(s) do Worklist = $n_{df}$+ Worklist
        forall statements $n_i$ not in any pair in Pairs do In[$n_i$ ]=Out[$n_i$ ]=$\emptyset$
        forall (s, v) $\varepsilon$ Pairs do In[s]= $\emptyset$;Out[s]={(s,v)}
        if Names then AffectedPreds ={$s_i$ }else AffectedPreds = $\emptyset$
        while Worklist $\neq \emptyset$ do
                Get n from head of Worklist
        NewIn = $\cup_{p\, \varepsilon Pred(n)}$ Out[ p]
        if NewIn $\neq$ In[n] then
                In[n]=NewIn
                PredCd = $\cup_{p\, \varepsilon Pred(n)}$ Cd( p)
        if PredCd - Cd(n) $\neq \emptyset$ then UpdateAffInfo
        if n has a c-use of variable v such that (d,v) $\varepsilon$ In[n] then
                forall (d,v) $\varepsilon$  In[n] do Triples = Triples $\cup$ {(d,n,v)}
                Kill ={(s,Def (n)): (s, Def (n)) $\varepsilon$In[n]}
                Out[n]=(In[n]-Kill) $\cup$ {(n, Def (n))}
        elsif n has a p-use of variable v such that (d,v) $\varepsilon$ In[n] then
                forall (d,v) $\varepsilon$In[n] do Triples = Triples $\cup$ {(d,n, p)}
                DefsOfV = BackwardWalk(n,{v}) - In[n]
                In[n]=In[n] $\cup$ {(n, $v_i$ ):(d,$v_i$ ) eDefsOfV}
                Out[n]=In[n]
                AffectedPreds = AffectedPreds $\cup$ {n}
        elseif n defines a variable v $\Lambda$ Cd(n) $\cap$ AffectedPreds $\neq \emptyset$then
                Out[n]=Out[n] $\cup$ {(n, Def (n))}
        else Out[n]=In[n]
                if Out[n] $\neq \emptyset$ then
                        forall x $\varepsilon$ Succ(n) do Worklist = $x_{df}$ + Worklist
        return(Triples)
end ForwardWalk
```

Figure 7: Algorithm *ForwardWalk* [14]

If the algorithm encounters a new definition of a variable *p* at statement n, then the values of *p* belonging to *In[n]*are killed by this definition, and the search for these values along this path terminates. The set *Kill* in the algorithm denotes the set of values killed by a definition. The *Kill* set is needed to compute *Out[n]*from *In[n]*. Since the algorithm walks forward in the control flow graph, *In*[*n*] is computed by taking the union of the *Out* sets of *n*'s predecessors. During this traversal, a work list, *Worklist*, consisting of those nodes that must be visited, indicates how far the traversal has progressed. *ForwardWalk* maintains *Worklist* as a priority queue based on a depth first ordering of nodes in the control flow graph. The algorithm examines the statements in *Worklist* for c-uses and p-uses of the definitions in the *In* sets along with definitions in statements that are control dependent on a changed or affected predicate. As the algorithm examines the statements in *Worklist*, it adds additional statements to be considered to *Worklist*. *ForwardWalk* also uses *DefsOfV*, a set of definitions, *v*, a program variable, and *k* and *n*, statements in the program. Functions *Pred*, *Succ* and *Def* return the predecessors, successors and variable defined by statement *i*, respectively.

```
procedure UpdateAffInfo
begin
forall $k \varepsilon (PredCd - Cd(n)) \cap AffectedPreds$ do
        $In[n] = In[n] - \{(k, v_i) \text{for all } v_i\}$
        $AffectedPreds = AffectedPreds - \{k\}$
        forall $(k, u, v)$inTriples do
                $Triples = Triples - \{(k,u,v)\}$
        forall $(d, v) \varepsilon DefsOfV$ do
                $Triples = Triples \cup \{(d, u, v)\}$
end UpdateAffInfo
```

Figure 8: Procedure UpdateAffinfo [14]

In the first part of *ForwardWalk*, all variables are initialized. The main part of the algorithm is a **while** loop that processes statements/nodes on *Worklist* until *Worklist* is empty. For each statement *n* removed from *Worklist*, processing consists of first computing *NewIn* for *n* by taking the union of the *Out* sets of the predecessors of n, and then determining if *NewIn* is the same as the previous value of *In[n]*.If these sets are the same, there has been no change since the previous iteration, and the forward walk along this path terminates at n. If these sets differ, *n* is processed further: *NewIn* is assigned to *In[n]*,and *PredCd* is assigned the union of the control dependence information for *n*'s predecessor(s). If *PredCd* contains nodes on which *n* is not control dependent, then the forward walk along this path has moved into a different region of control dependence, and *AffectedPreds* must be updated accordingly.

Procedure *UpdateAffInfo* shown in Figure 8 handles this task. Then, node *n* is checked to see if it has a c-use of variable *v*. If so, def-use associations for any pairs in *In[n]* are added to *Triples*, and the variable defined at *n* is added to *Out[n]* since it is indirectly affected. If there is no c-use of *v* at *n* but *n* contains a p-use, def-use associations for any pairs in *In[n]* are added to *Triples*. Since a p-use signals an affected predicate, any definitions that reach *n* is found using *BackwardWalk*; these definitions are used to identify indirectly affected def-use associations.

If neither type of use is found at *n*, the statement is inspected for a definition; if one is found at *n*, the appropriate data flow sets are updated. Finally, if no definition or use of the variables is found at *n*, the data flow information is propagated through n. If *Out[n]* is not empty, then the successors of *n* must be processed, and they are added to *Worklist*. When *Worklist* is empty, processing terminates and *Triples* is returned.

2.2.3 Advantages and disadvantages of static program slicing

Static program slicing [57] derives its information through the analysis of the source code. Its strength can be found particularly in the following areas:

(a) The computation of a static program slice is relatively cheap (compared to the dynamic program slice) as only the static analysis of the source code and no analysis of program execution is required

(b) It helps the user to gain a general understanding of the program parts that contribute to the computation of a selected function with respect to all possible program executions.

(c) No operational profile required.

However, static program slicing has some major drawbacks and they are as follows:

(a) For programs containing conditional statements, dynamic language constructs like polymorphism, pointers, aliases, etc., static slicing has to make conservative assumptions with respect to their run-time contribution that might be relevant for the slice computation.

(b) Due to its static nature, static program slicing does not provide any information with respect to the analysis of program executions as slices are based on static information.

(c) In most cases, static program slicing produces larger program slices than the dynamic program slicing algorithms.

2.3 Dynamic program slicing

The goal of program slicing is to find the slice with the minimal number of statements but this goal may not be always achievable in general static program slicing. A dynamic program slice overcomes the limitations of the static program slicing algorithms as it is based on a particular program execution (program input). A dynamic program slice, as originated by Korel and Laski [23], is an executable part of the program whose behavior is identical, for the same program input, to that of the original program with respect to a variable of interest at some execution position. In the existing dynamic program slice algorithms the major goal is to identify those actions in the execution trace that contribute to the computation of the value of variable y^q by identifying data and control dependencies in the execution trace. However, it is also important to identify actions that do not contribute to the computation of y^q. The more such "non-contributing" actions that can be identified, the smaller will be the dynamic program slice computed by the algorithm. A slicing criterion of program P executed on program input x is a tuple $C=(x,y^q)$ where y^q is a variable at execution position q. A *dynamic program slice* of program P on slicing criterion C is any syntactically correct and executable program P' that is obtained from P by deleting zero or more statements. In addition, the *dynamic program slice* when executed on program input x produces an execution trace T'_x for which there exists a corresponding execution position q' such that the value of y^q in T_x equals the value of $y^{q'}$ in T'_x. A dynamic program slice P' preserves the value of y for a given program input x. The goal to find the smallest slice may be difficult, however, it is possible to determine a safe approximation of the dynamic program slice that will preserve the computation of the values of variables of interest. Most of the existing algorithms of dynamic program slice computation use the notion of data and control dependencies to compute dynamic program slices. Dynamic program slicing algorithms presented in [3,13] do not compute correct slices for unstructured programs (shown in [30]) and/or procedural language constructs. In [21] an algorithm for the computation of inter-procedural slicing of structured programs was presented. However, this algorithm is limited to structured programming language constructs. In [30] a dynamic program slicing algorithm based on the notion of removable blocks was introduced. This algorithm computed correct executable slices for unstructured non-object-oriented programming languages. Later, the algorithm was further refined in [45] for all language constructs found in major procedural programming languages. A forward computation of dynamic slices for structured programs was introduced in [28] that does not require the recording of an execution trace. A dynamic program slicing for object-oriented programs based on forward computation was introduced in [59] that computes non-executable program slices but it is not based on the notion of removable blocks.

By *last definition LD* (v^k) of variable v^k in execution trace T_x , it means, action Y^p such that (1) $v \in D(Y^p)$ and (2) for all i, $p < i < k$ and all Z such that T_x (i)=Z, $v \in D(Z^i)$, in other words action Y^p assigns a value to variable v and v is not modified between positions p and k. For example, the last definition of variable *Showroom_price* at node 12^9 in execution trace of Figure 2 is action 9^5.

Dynamic data dependence captures the situation where one action assigns a value to an item of data and the other action uses that value. For example, in the execution trace of Figure 2, 3^3 assign a value to variable *MSRP* and 4^4 uses that value.

Dynamic control dependence captures the influence between "test" actions and actions that have been chosen to be executed by these "test" actions. The concept of control dependence may also be extended to actions by using the concept of control dependence between nodes. Action Z^k is control dependent on action Y^p iff (1) $p < k$, (2) Z is control dependent on Y, and (3) for all actions X^i between Y^p and Z^k, $p < i < k$, X is control dependent on Y. For example, action 9^6 is control dependent on action 4^4 as action 8^5 is control dependent to 4^4 in the execution of Figure 2. Figure 9 shows a dynamic program slice for variable *Showroom_price* at node 12, with input *MSRP* = 25000. A dynamic program slice can be regarded as a refinement of the static program slice. By applying dynamic analysis [23] it is easier to identify those statements in the program, that does not have influence on the variables of interest.

```
1^1.    normal_profit = 100;
3^2.    cin>>MSRP;
4^3.    if (MSRP > 30000 )
        {
         }
8^4.    else
        {
9^5       Showroom_price := MSRP + normal_profit;
        }
12^6   cout<<Showroom_price;
```

Figure 9: Dynamic slice for *Showroom_price* at node 12,
with input MSRP = 25000

2.3.1 Dynamic backward program slicing algorithm

The dynamic backward program slicing algorithm as presented by Korel [23] is shown in Figure 10. In the first step (line 1) of the algorithm, a program P is executed on input x and the execution is recorded up to execution position q. In step 2, all nodes in the execution trace are set to unmarked and not visited. The third step finds the last definition of the variable y^p and sets it as marked. On each step of the **while** loop (4-10) a marked and not visited action X^k is selected and set as visited (lines 5 and 6). Line 7 that corresponds to finding data dependencies between actions, all variables used in X^k are identified and marked. Line 8 corresponds to finding the control dependencies for the node X^k and marks existing control dependencies on action X^k. In line 9, all multiple occurrences of action X in the execution trace are marked. The while loop iterates until all marked actions is visited.

Korel's Algorithm:

Input: a slicing criterion $C=(x,y^q)$;
T_X be an execution trace up to execution position q.

Output: a dynamic program slice of variable y at position q

```
1.  Execute program $P$ on input $x$ and record execution trace $T_X$ up to position $q$
2.  Set all nodes in $T_X$ as unmarked and not visited
3.  Find last definition of $y^p$ and set $y^p$ as marked
4.  While there exists a marked and not visited action $X^k$ in $T_X$ do
5.      Select a marked but not visited action $X^k$ in $T_X$
6.      Set $X^k$ as visited
7.      For all variables $v \in U(X^p)$ do Find and mark last definition $v^p$ of $v$
8.      Mark all actions $Z^t$ such that there exists a control dependence between $Z^t$ and $X^k$
9.      Mark all multiple occurrences (actions) of node $X$.
10  End-While
11. Show a dynamic program slice that is constructed from $P$ by removing nodes
    (statements) whose actions were not marked in $T_X$.
```

Figure 10: Backward algorithm for computing dynamic slices[23]

2.3.2 Dynamic forward program slicing algorithm

The main motivation for developing the forward approach of computing dynamic slices was to overcome the execution trace recording in the backward algorithm approach. The forward computation of dynamic slices was proposed by Korel and Yalamanchili [28]. In the approach, a dynamic program slice is computed during program execution on input *x* and no major execution trace recording is required. The underlying idea of the forward approach of dynamic program slice computation is that during program execution on each exit from a block the algorithm determines whether the executed block should be included in a dynamic program slice or not.

The necessities to record the execution trace during program execution that results in its limited usability for relatively short executions. The idea of finding dynamic program slices is based on the notion of removable blocks. The forward algorithm computes a dynamic program slice for every program variable defined during the program execution for input *x*. The forward algorithm starts from the first node in the program and proceeds "forward" with program execution and at the same time perform the computation of the dynamic slices for program variables along with the program execution.

The forward approach uses the notion of a *NODESLICE(X)* which is similar to the notion of dynamic program slice *SLICE(v)* with the major difference that *NODESLICE(X)* preserves the behavior of all executions of a node whereas *Slice (v)* preserves the values of variables.

The following two conditions describe the general rules under which the executed blocks may not be included in dynamic slices.

- X^k is an action corresponding to the execution of a simple block. If X is not included in the dynamic program slice for the variable before the execution of X^k and v is not defined in the node X^k, then the node X does not have to be included in the slice, since variable v is not modified during the execution of node X^k and X does contribute to the computation of the value of v.
- If the value of variable v has not been modified in block B, and for none of the nodes executed inside of block B, and B does not belong to the Slice for that specific variable after leaving the block, the slice for the variable remains unchanged from the beginning of the block, hence it does not contribute to the computation of the value of v at the exit of the block.

The forward slicing algorithm presented in [28] is shown in Figure 11. The following is a list of all major data structures used in the forward computation of dynamic slices.

SLICE (v)	*Slice*(k,{v}), is a dynamic program slice of variable v at the current execution position k.
NODESLICE (X)	is equivalent to *NodeSlice*(k,X) at the current execution k
B	is a block id.
BL	is a stack of blocks inside of which the program is being currently executed.
BV(B)	is a set of variables modified during the current execution of block B.
TopSlice(B,v)	contains a dynamic program slice of v at the entry to block B.
BlockFlag(B,v)	is marked if during the execution of block B a node belonging to *TopSlice*(B,v) is executed.

```
Execute program P on input x. On entry node do:
        for all $v \in V$ do $SLICE(v) := \varnothing$;
        for all $X \in N$ do $NODESLICE(X) := \{X\}$;

Node $X^k$: On each execution of node $X^k$ the following steps are performed
1. Action $X^k$
a.
$NODESLICE(X) := NODESLICE(X) \cup SLICE(v), v \in U(X^k)$
b.
for all $v \in V$ do
        if $v \in D(X^k)$ then
                        $SLICE(v) := NODESLICE(X)$
                        for all $B$ on $BL$ do $BV(B) := BV(B) \cup \{v\}$
        else
                        if $X \in SLICE(v))$ or ($X$ is not simple block)
                        then $SLICE(v) := SLICE(v) \cup NODESLICE(X)$
        end if
        for all $B$ on $BL$ do
                        if $X \in TopSlice(B,v)$ then
                                BlockFlag(B,v)
        end for
end for
2. Entry into block $B$
        $BL$ := Push $B$ on $BL$
        $BV(B) := \varnothing$
        for all $v \in V$ do
                        $TopSlice(B,v) := SLICE(v)$
                        $BlockFlag(B,v)$ := unmarked
        end for
3. Exit from block $B$
        $BL$ := Pop $B$ off $BL$
        for all $v \in V$ do
                        if ($v \notin BV(B)$) and ($BlockFlag(B,v)$ = unmarked)
                        then $SLICE(v) := TopSlice(B,v)$
        end for

4. $k = q$ (execution reaches position $q$);
Display $SLICE(y)$
```

Figure 11: Forward algorithm for dynamic program slicing[28]

When program *P* is executed on some input *x* at each step of its execution, the forward algorithm computes dynamic slices for all program variables. For this purpose, at the execution of each action X^k, the following steps are performed.

In step 1a (line 8,9), *NODESLICE(X)* is computed for the currently executed node *X*. In Step 1b, the dynamic slice for each variable *v* in the program is created, resulting in two cases which must be considered:

1. If variable *v* is defined at *X*, then the slice of *v* is given (in line 13) by *SLICE(v) = NODESLICE(X).*
2. If *X* is not a simple block (e.g. a test node) or if *X* already belongs to dynamic program slice *SLICE(v)* then the slice is computed in line 17.

Finally, in step 1b, a data structure, *BV(B)* is used to store all the variables that are modified in the current block *B*; this is used later in step 3. In addition, the algorithm checks as to whether or not the currently executed node *X* belongs to the dynamic program slice of each program variable at the top of the current block; if the node *X* belongs to the dynamic program slice then the algorithm sets a corresponding entry in data structure *BlockFlag(B,v)* as marked.

At each entry to block *B* step 2 is performed. In step 2, the algorithm is storing currently computed dynamic slices in *TopSlice(B,v)* for each program variable *v*. In addition a set *BL* of currently executed blocks is maintained. On the exit from block *B*, step 3 is executed and the algorithm checks the following condition for each program variable *v*. If during the current execution of block *B*, variable *v* is not defined or *BlockFlag (B,v)* is unmarked, then the dynamic program slice of *v* is set to *TopSlice(v).* Finally when execution reaches the execution position *q* the algorithm terminates and displays dynamic slice *SLICE(y)* in step 4.

2.3.3 Advantages and disadvantages of dynamic program slicing

As already stated, dynamic program slicing is further refinement of static program slicing, the following are considered main advantages as compared to the later:

(a) Dynamic program slicing allows a reduction in the slice size and a more precise handling of arrays and pointer variables at runtime.

(b) Dynamic program slicing computation can utilize information about the actual program flow for a particular program execution, which leads to an accurate handling of dynamic and conditional language constructs and therefore leads to smaller program slices.

(c) Allows for additional application in performance analysis and debugging.

There are few associated disadvantages to get the above benefits and they are:

(a) In dynamic program slicing (compared with static slicing), it is necessary to identify relevant input conditions for which a dynamic program slice should be computed. A commonly used approach to identify such input conditions is referred to as an operational profile, which is a well-known concept that is frequently applied in testing and software quality assurance.

(b) The computation of dynamic slices is based on a particular program execution that incurs a high run time overhead due to the required recording of program executions and/or analysis of every executed statement.

2.4 Hybrid program slicing

Hybrid program slicing algorithms takes advantage of the best properties of both static and dynamic slicing to derive a compromising slicing solution. However, very few research works focus on this type of slicing. Gupta and Souffa in [15] used pre-set breakpoints history information in their static slicing to solve conditional predicates. The procedure is characterized by them as follows:

- The user sets the breakpoints and starts the execution.
- When the breakpoint is encountered, the user examines the values of variables at breakpoint.
- If the values are as expected, the user resumes the program execution. However, before resuming the execution the user may disable some breakpoints or add new ones.
- If the values are incorrect, the user requests slicing information for selected variables to potential cause of error.

Information on conditional predicates helped to reduce the size of the static program slice without having to generate complete execution trace of the program. It assumes user can identify the breakpoints at various points in the program. Limitations of this approach are that it only supported structured programs, assumes user's knowledge of program for breakpoints and it might not compute executable slice. It is generally agreed that mixing of static and dynamic program slicing is a good compromise between accuracy and time performances. However, the user should interfere as little as possible to compute the slice [47]. Gupta et al, in their paper [15], have presented an algorithm based on breakpoint history and some experimental results of their work. Schoenig and Ducass'e [47] hybrid backward slicing algorithm for *Prolog* and is only applicable to a limited subset of *Prolog* programs. They claimed only preliminary prototype and there is no further research evidence on their algorithm or test results.

2.4.1 Comparison of hybrid program slicing with other program slicing approaches

Dynamic program slicing algorithms have advantage with respect to their accuracy in handling dynamic language constructs. However, the computation of dynamic slices is based on a particular program execution that incurs a high run time overhead due to the required recording of program executions and/or analysis of every executed statement. The time and space required for recording execution trace and traversing the same depend on the number of times each of these executed statements might be significant. For example, the if the program has a "**for**" loop for **n** times and has **m** statements in the loop, then the **m** x **n** entries have to be stored/traversed and analyzed. Many statements are executed merely because they are part of the program flow but their execution might not be relevant at all for the computation of the selected function. In the above example, for instance, say there is only **l** statements that are relevant to the computation of the selected function. This will result in unnecessary tracing and traversing of ((**m-l**)+1) x **n** executed statements during slice execution.

As the source code size grows, space and time complexity increases drastically for the dynamic program slice computation due to number of executed statements that have to be stored and analyzed.

Static program slicing, on the other hand, only analyzes the source code and requires therefore less of an overhead during the slice computation. As already known it computes conservative program slice which is an undesirable property. At the same time, the cost computation is far cheaper than the dynamic program slicing.

In nutshell, hybrid program slicing shall use the cheaper computation property of static program slice as in [15] but should improve the accuracy of the slice with some incremental cost. However, the algorithm presented in [15] is clearly not the choice for larger programs as it is impractical set the breakpoints for every conditional statement unless user has very good comprehension of the code.

Definitely, the quality of the slice should be progressed towards dynamic program slice. To achieve better slicing algorithms, one of the logical steps is to give the user some choices to choose between size and accuracy. Other choice is to reduce the size of the execution trace recording in dynamic program slicing by some means. One of the ways to reduce the recording trace is to use the condensed program such that unwanted statements are executed in the first place. Other way is to suspend the execution trace recording at a given criterion.

In the current research, an attempt is made to get the above properties to enhance the program slicing algorithms and is detailed in section 4.

2.4.2 Advantages and disadvantages of hybrid program slicing

As stated earlier, the hybrid program slicing uses the properties of both static and dynamic program slicing, the combination of the two yields certain advantages and they are:

(a) It allows for reduction of the space and time for the computation as it keeps out the non-specific statements as compared to pure dynamic program slicing.

(b) It will give more precise slice than a static program slice, if not it is same as static program slice.

(c) It will help the programmer to carry out multiple executions at a low cost with various values for the same variable to understand behavior of the source code.

However, to bring the two major program slicing techniques together, hybrid program slicing has to carry out additional work and it poses a few disadvantages and they are:

(a) Additional run time required as compared to static program slice if the programs relatively small as in hybrid program slicing use both static and dynamic slicing algorithms.

(b) Not all the hybrid program slicing guarantees the accuracy of dynamic slicing as it depends on the hybrid program slicing algorithm and criterion.

This section presents the critical review of the existing program slicing techniques that forms the foundation of this thesis. The review shows that the previous algorithms require further refinement and new additional concepts based on notion of removable blocks required for better usability of program slicing techniques.

3. Static program slicing based on removable blocks

The general static program slicing algorithm introduced in this section computes correct program slices for all language constructs found in major object-oriented programming languages. Existing static program slicing approaches rely upon a collection of data flow equations and related control flow graphs(CFG). Other approaches use program dependence graph (PDG) combined with some variations of a syntax tree by essentially using nodes and their edges to represent control and data dependencies between nodes [16]. Static program slicing algorithms that derive their information through static inspection of the source code were originally introduced for imperative procedural programming languages. This type of technique may be inadequate for object oriented programming constructs (e.g.: inheritance, dynamic binding, polymorphism, etc.). Therefore, have to make conservative assumptions with respect to dynamic language constructs. A number of approaches have been proposed for to extend static slicing for object programs [6,34,36,37,54,59].

In this section, a new general static program slicing algorithm is presented and it extends the notion of removable blocks applied for dynamic program slicing [28]. In the new static program slicing algorithm, a further refining of the static slicing is used. This extended notation of removable blocks for object-oriented programming language constructs is explained below and it applies for the general static program slicing algorithm.

3.1 Extended notation of removable blocks

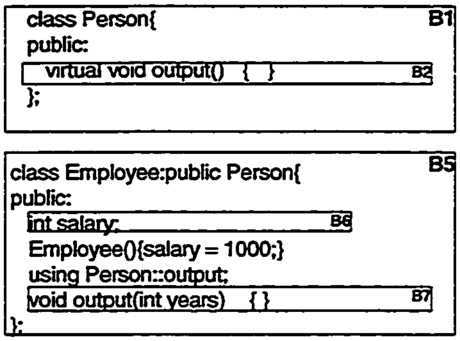
The following are extensions and definitions of removable blocks that are introduced to provide support for object-oriented programming constructs:

Dynamic Binding/Polymorphism

In object-oriented programs, messages are sent to objects instead of calling procedures. Heterogeneous sets of objects can be treated uniformly at the sender side while appropriate reaction on the receiver side is guaranteed. Polymorphism and dynamic bindings provide much of the power of object-oriented programming and at the same represent (in particular) for static algorithms a major challenge. A static analysis of dynamic binding/polymorphism was presented in [52]. For the general static program slicing algorithm presented in this research, classes and their associated member functions are treated as regular removable blocks based on the original definition of removable blocks. The only exception as part of this definition is that both, the "constructor" and "destructor" within a class are treated as non-removable blocks see block B5 in Figure 12, to ensure the computation of executable program slices.

3.1.1 Slice computation in the presence of inheritance and template classes

Inheritance allows objects to derive attributes and behavior from their parent classes. A base class that do not have an instance can only be removed if none of the inherited classes or any statements in these inherited classes are included in the program slice.



```
class Person{
public:
  virtual void output()  { }
};

class Employee:public Person{
public:
  int salary;
  Employee(){salary = 1000;}
  using Person::output;
  void output(int years)  { }
};
```

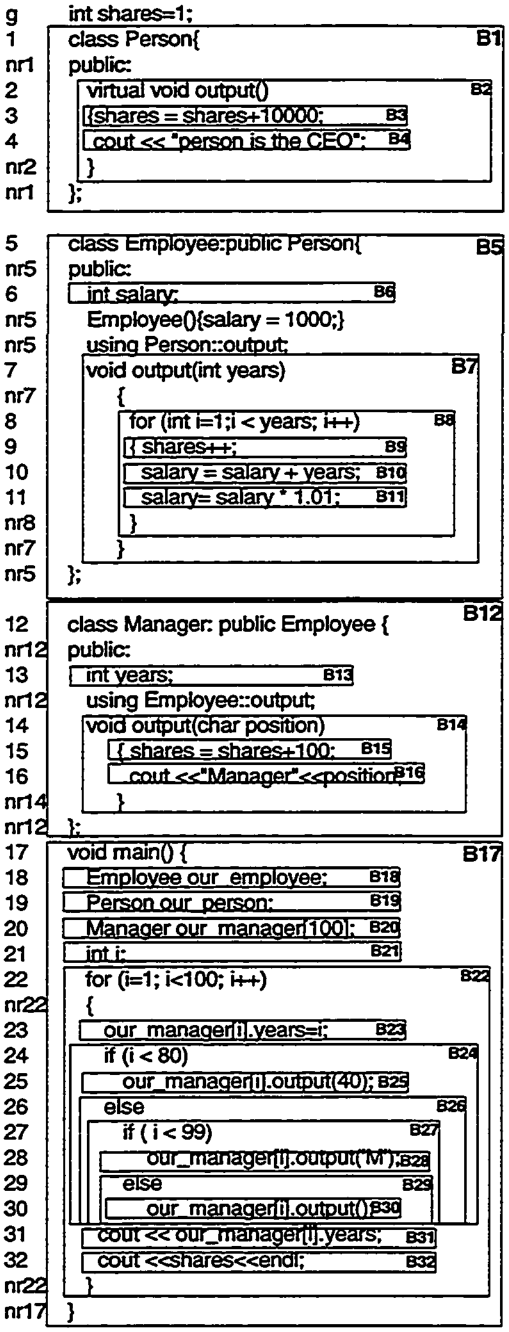
Figure 12: Block representation of an OO program

Figure 12 shows clearly that if block *B5* (Employee) is part of the slice (not removable) then the "*constructor*", "*destructor*" (implicit or explicit) and associated class construct like "*using*" with this class cannot be removed. The inherited class has also to be included in the program slice (represented by block *B1*) to ensure an executable program slice. The same rules will apply for the computation of slices for the templates.

3.2 Static program slicing algorithm

The general static program slicing algorithm is based on the notion of removable blocks that was originally introduced in Korel's [28] dynamic program slicing algorithm. Korel's dynamic program slicing algorithm [30] is shown in appendix A, and it includes theories that are applied to derive general static program slicing algorithm.

The hypothesized static program slicing algorithm differs from many classical approaches of static program slicing by combining the notion of a syntax tree with removable blocks. At the same time, it shares the values of detection slices in line with Weiser [57] and others. The syntax tree with removable block concept is new to static program slicing and it improves the visualization properties as compared to data flow graph based representation. The underlying principle of the static program slicing algorithm is based on the assumption that each statement corresponds to a block that is either contributing, non-contributing or neutral.



```
g     int shares=1;
1     class Person{                                   B1
nr1   public:
2       virtual void output()                         B2
3       {shares = shares+10000;           B3
4        cout << "person is the CEO";     B4
nr2     }
nr1   };

5     class Employee:public Person{                   B5
nr5   public:
6       int salary;                       B6
nr5     Employee(){salary = 1000;}
nr5     using Person::output;
7       void output(int years)                        B7
nr7       {
8          for (int i=1;i < years; i++)               B8
9          { shares++;                    B9
10           salary = salary + years;     B10
11           salary= salary * 1.01;       B11
nr8        }
nr7       }
nr5   };

12    class Manager: public Employee {                B12
nr12  public:
13      int years;                        B13
nr12    using Employee::output;
14      void output(char position)                    B14
15        { shares = shares+100;          B15
16          cout <<"Manager"<<position;   B16
nr14      }
nr12  };
17    void main() {                                   B17
18      Employee our_employee;            B18
19      Person our_person;                B19
20      Manager our_manager[100];         B20
21      int i;                            B21
22      for (i=1; i<100; i++)                         B22
nr22    {
23        our_manager[i].years=i;         B23
24        if (i < 80)                                 B24
25          our_manager[i].output(40);    B25
26        else                                        B26
27          if ( i < 99)                              B27
28            our_manager[i].output('M'); B28
29          else                                      B29
30            our_manager[i].output()     B30
31        cout << our_manager[i].years;   B31
32        cout <<shares<<endl;            B32
nr22    }
nr17  }
```

Figure 13: A Sample program with removable blocks

A sample program with removable blocks is shown in Figure 13. The tree structure of the blocks follows the syntax flow in question rather than control and data dependencies. The new syntax tree representation for the sample program is shown in Figure 14. As already stated, the algorithm is based on the notion of removable blocks, where each removable block can be either a simple or a complex block. Each function or class is represented using a new syntax tree (as shown in Figure 14). This new approach simplifies the syntax diagram by reducing essentially into blocks encompassing the nodes that are considered removable in the beginning. By utilizing the notion of removable blocks organized in the presented tree structure, the identification of the scope of a particular block (statement) is made clear.

The organization of the removable blocks with syntax tree structure (Figure 14 and Figure 15) also simplifies the navigation and the identification of the scope of a particular block (statement). If any inner block is considered as non-removable, then automatically all outer blocks encompassing the inner block are considered non-removable where as in other approaches (e.g. CFG), one has to traverse over the edges to find the relevant parent nodes. For example, consider block 30 to be included in the slice, one has to include {29,27,26, 24,22,17} as well as the blocks they depend for data (Figure 14). In the current approach, a system dependence graph (SDG)-like principle is used for handling procedure-calling contexts to jump to the relevant syntax tree and later follow the general approach presented above.

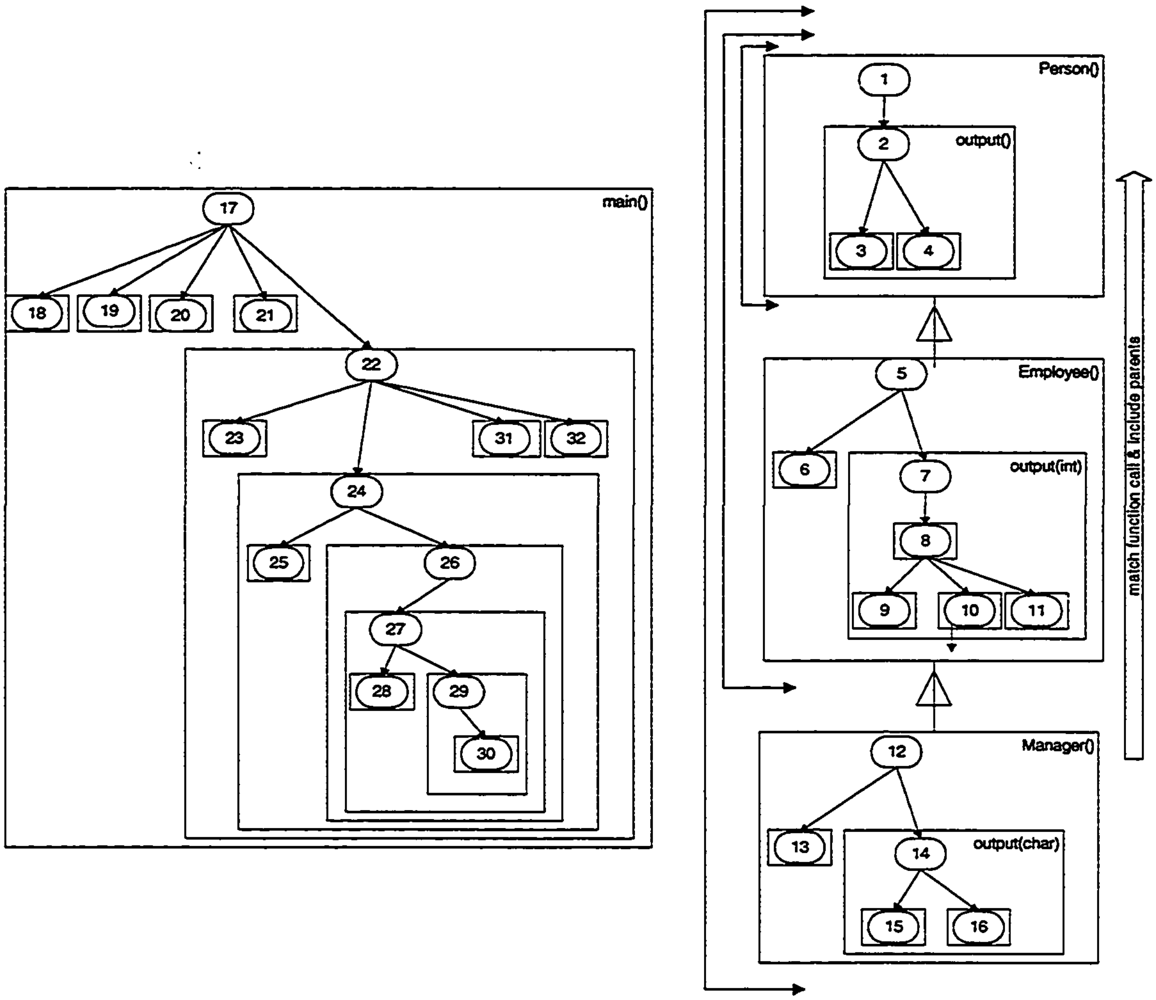


Figure 14: Illustration static syntax tree with removable blocks.

3.2.1 Static syntax tree with removable blocks

As a pre-requisite for the static program slicing, the source code statement information is extracted using general MOOSE parser. Using the information derived by the parser, one or more syntax trees are created and stored in the memory. Each free function and class will have a separate tree structure as shown in Figure 14. All the syntax trees are known to the framework and it can be used to for reaching for a particular tree during the navigation. Each tree starts with a root block and is usually the function or class declaration. Each removable block sequentially takes its place in the tree either as child or sibling depending on its reachability from root. It should be noted that always the last *removable block* is the right most block of the tree and similarly first block (root block) is the left most block.

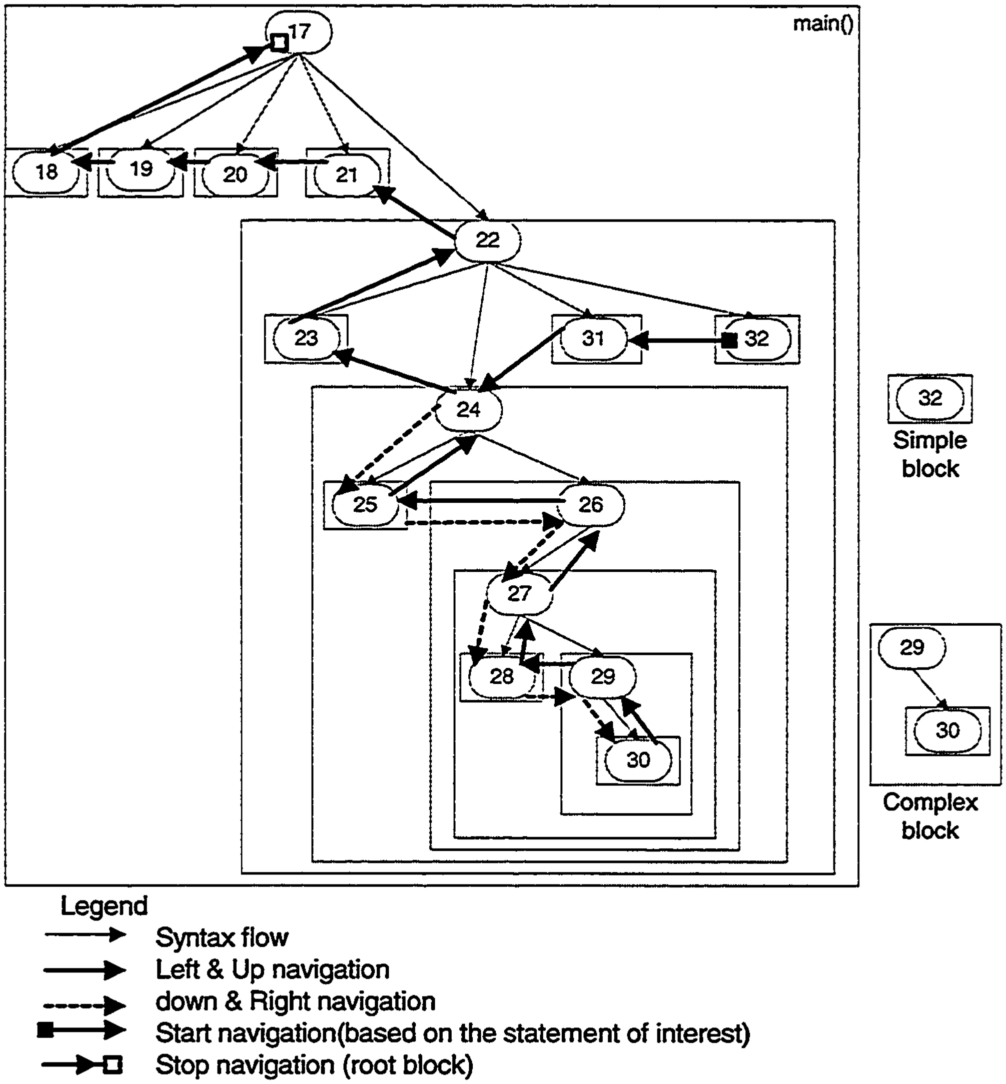


Figure 15: Illustration of static syntax tree navigation.

As explained earlier, inherited classes and their methods may become non-removable based on their instances. For example if there is no instance of *class Manager,* then class and its methods are not included in the resulted slice for particular slicing criterion. On contrary, if there is an instance of *class Manager* and no other parental class instances, as mentioned earlier, all the parental class body that is neutral (Example: constructor, class declaration etc.) would be included.

The hierarchy of inclusion is shown by an upward arrow in Figure 14. Figure 16, shows the removable blocks after algorithm's navigation and evaluating for *"shares"* at statement 32. It is clear that the algorithm includes the entire "test" blocks or neutral blocks that form the hierarchy. The blocks {29,27,26,24,22,17} are considered non-removable blocks. For example, say only block 30 is required by *definition* and *use* algorithms, then all the blocks encompass block 30 are included in the slice. The inclusion and exclusion of a block is determined by its relative position to the other blocks. Any inner block inclusion will automatically include all outer control or predicate blocks.

3.2.2 Static syntax tree navigation

One of the most important properties of this new tree approach is its navigation technique where each block can be navigated without requiring any duplicated efforts to check whether a particular block is removable or not. Two important navigation principles are used to navigate through the tree: *"left & up"* and *"down & right"*. The *"left & up"* technique traverses the tree from "right" to "left" and "down" to "top". The *"down & right"* technique traverses the tree from top to down and left to right. It should be noted that both techniques applied recursively, always carry out the complete navigation from last leaf block to root block. Figure 15 illustrates the navigation methods used in the algorithm.

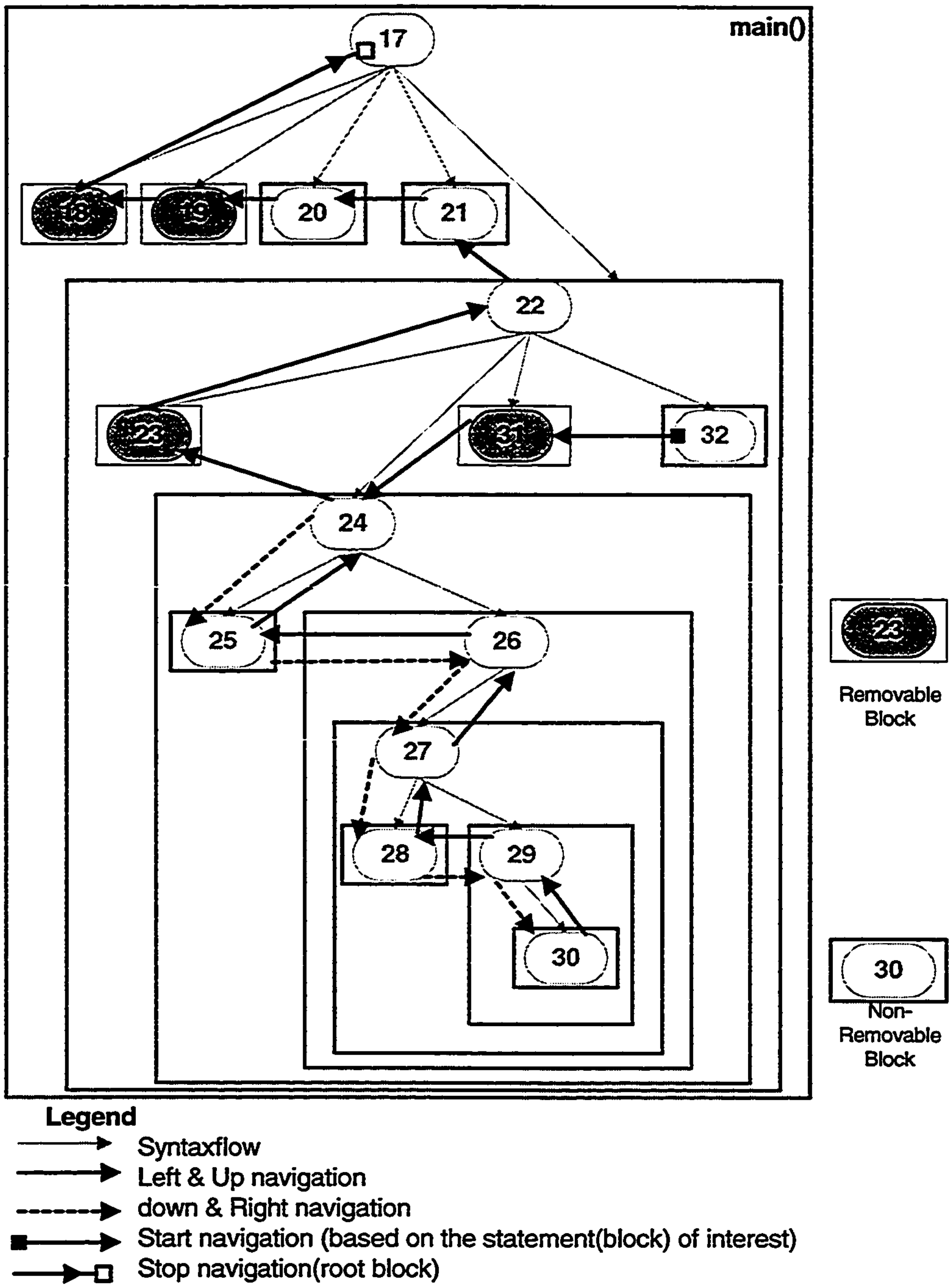


Figure 16: Illustration syntax tree after navigation for *"shares"* at statement 32

3.2.3 Similarities with Korel's algorithm

The static program slicing algorithm takes advantage of several properties: (1) The algorithm utilizes the same notion and definitions of removable blocks as the general dynamic program slicing algorithm introduced by Korel [30] which is shown in appendix A. (2) The same proof of correctness holds for the general static and dynamic algorithm since both are based on the same notions and principles. (3) The internal representation (syntax tree and removable blocks) and navigation used in the algorithm is easy to derive and to implement and it is an enhanced block tree concept presented in [30]. (4) The algorithm computes executable static slices for all program language constructs. It should be noted that the algorithm applies a conservative approach towards the handling of dynamic language constructs and might be therefore compute larger (less accurate) program slices than the dynamic algorithm. Similarly, to the problems described in Steindl [52], the presented algorithm will handle polymorphism and dynamic binding by including all possible candidates with respect to the dynamic binding/polymorph functions in the static program slice, without requiring any user interaction.

3.2.4 Definitions of static program slicing algorithm

For the general static programming algorithm, the notion of a tree trace is introduced that can be regarded as a variation of the block trace definition introduced in [30]. A tree trace T_{Trace} $(B, p_1, p,)$ of block B that belongs to R_{bl} by finding *r-entry* and *r-exit* of B at position p_1, and p where p_1 and p represent tree position based on the applied tree navigation. General static program slicing algorithm based on removable blocks is shown in Figure 17.

3.2.5 *Description of algorithm*

The algorithm identifies a set of removable blocks R_{bl}. In the first step of the algorithm, the program is initialized with tree trace Tr_{bl} and is stored in memory. All blocks in the tree trace Tr_{bl} are set as unmarked and not visited . Set R_{bl} initially contains a set of all blocks in the program. In step 3, a set of contributing blocks I_{bl} is initialized as an empty set. In step 4 the algorithm identifies and marks the last definition Y^{pbl} of variable y^{sbl}, this step (procedure) is presented in more detail in lines 47-51. The algorithm iterates in the repeat loop 5-8 until all marked blocks are visited. There are two major steps inside of the repeat loop.

In step 6, the algorithm identifies blocks that contribute to the computation of y^{sbl}. In step 7, for the given set of contributing blocks, the algorithm identifies non-contributing blocks by finding a set of tree traces. These blocks are visited on the next iteration of the repeat loop. The process of identifying contributing and non-contributing blocks continues in the repeat loop until all blocks are classified as either contributing or non-contributing actions.

In the following paragraphs, a more detailed description of the major steps of the static program slicing algorithm is presented. In step 6, the algorithm identifies contributing blocks. This step (procedure) is presented in more detail in lines 10-22. The major component of this "while" loop is that during each iteration a marked and not visited block X^{bl} is selected and set as visited in line# 13. In addition, X^{bl} is inserted into I_{bl} in line# 14. All last definitions of all variables used in $X^{bl}$are identified and marked in line#15-17. In line# 18 -20, all blocks that contain node X are removed from R_{bl}. The "while" loop iterates until all the marked blocks are visited. In step 7 the algorithm identifies non-contributing blocks for the set I_{bl} of contributing blocks (the blocks are set as visited) and for the set of blocks R_{bl}. This step (procedure) is presented in more detail in lines 23-41. The goal of this step is to find as many non-contributing blocks as possible. The procedure identifies non-contributing blocks by finding a set Φ_{bl} of tree traces for the set of blocks R_{bl}. All tree traces of Φ_{bl} contain only unmarked blocks. The procedure explores the tree trace from the s looking for blocks that are not set as marked. If such an block is found, then the procedure tries to identify tree trace T_{Trace} (B, pbl_1, pbl) of block B that belongs to R_{bl} by finding r-entry and r-exit of B at position pbl_1 and pbl, respectively. If such a tree trace is found then it is inserted into Φ_{bl}. One may notice that there are two main differences between static and dynamic slicing algorithms [30]. The first difference is that there is no identification and marking of the neutral blocks method in the repeat loop. This method has been replaced by *Identify-parent-declarations* to identify the parents and declarations that are explained detail in lines 52-55. The second difference is non-contributing procedure starts from the last visited block than starting from the root. Since all blocks in tree trace T_{Trace} (B, pbl_1, pbl) are non-contributing blocks, the algorithm continues the search for non-contributing blocks starting from position pbl_1. This process of identifying tree traces of blocks that belong to R_{bl} continues until the tree trace has reached root position of a particular syntax tree.

Similar to the generic dynamic program slicing algorithm, the identification of the last variable definitions and their scopes and the identification of inherited classes are also included in the current static program slicing algorithm. As part of the object-oriented languages constructs, additional variable types and their variable scopes have to be considered. The function in line# 47-51 not only identifies the scope of variables, it also has to take into consideration the variable types, e.g.; user defined, passed by value, passed by reference, etc. The function *identify_inheritance* in line# 42-46 handles the identification of inheritance and multiple inheritance cases as described earlier, by marking all classes from which the current class inherits as non-removable.

Algorithm

Input: a slicing criterion $C=(y^{sbl})$
Output: a static program slice for C
Legend:

Tr_{bl}:	Tree with *bl* blocks
R_{bl}:	Set of blocks for program *P*
Φ_{bl}:	Set of non contributing blocks
I_{bl}:	Set of contributing blocks (default is visited)
N_{bl}	non-contributing blocks
C_{bl}	contributing blocks
X^{bl}	current evaluated block position
y^{sbl}	Variable y at simple block s

⟲ Loops recursively

```
Initialize $R_{bl}$ to a set of all blocks in program $P$
Set all blocks in $Tr_{bl}$ as not marked and not visited.
$I_{bl} = \emptyset$
Find last definition $y^{pbl}$ of variable $y^{sbl}$ and set $y^{pbl}$ as marked
repeat
        Identify $C_{bl}$
        Identify $N_{bl}$
until every marked block in $Tr_{bl}$ is visited
Create static program slice by removing all blocks $\in R_{bl}$
function Identify $C_{bl}$
while (contributing and not visited block in $Tr_{bl}$)
    Select a contributing and not visited block $X^{bl}$ in $Tr_{bl}$
    Set $X^{bl}$ as a visited block & Identify-parent-declarations, Identify-inheritance
    $I_{bl} = I_{bl} \cup \{ X^{bl} \}$
    for all variables $v \in U(X^{bl})$ do
        Identify + mark last definition of $v$ as contributing block
    end-for
    for all blocks $B \in R_{bl}$ do
        if $X^{bl} \in N(B)$ then $R_{bl} = R_{bl} - \{B\}$
    end-for
end-while
end Identify $C_{bl}$
```

Figure 17: General static program slicing algorithm(continued)

```
function Identify $N_{bl}$
$\Phi_{bl} = \emptyset$
$pbl$ = last marked and visited block
do
    Let $X^{pbl}$ be a block at position $pbl$ in $Tr_{bl}$
  if $X^{pbl} \notin I_{bl}$ then
    $y^{tbl} \in I_{bl}$ be the closest $C_{bl}$ from position $pbl$ by navigating
     the syntax tree (left & up/down & right) such that $tbl<pbl$
     if there exists block $B \in R_{bl}$ such that
          (1) $B$ has an r-exit at position $pbl$, and
          (2) there exists no inheritance and function call
          (3) B has an r-entry between $tbl$ and $pbl$ then
     Let $pbl_1$ be the closest position of an r-entry of $B$ from position $pbl$, where
     $tbl<pbl_1<pbl$
         $\Phi_{bl} = \Phi_{bl} \cup \{ T_{Trace} (B, pbl_1, pbl )\})$
           $pbl_1 = pbl - 1$ by navigating the syntax tree (left & up/down & right)
       end-if
   end-if
  $pbl := pbl - 1$
while $pbl != 1$
end Identify $N_{bl}$
function Identify inheritance
if current class is derived then
    set $X^{pbl}$ as contributing + move left & up to mark parent class
end-if
end Identify-inheritance
function Find last definition
  Find last definition of variable $y^{pbl}$ in global scope
  If not found last definition then
     search last definition of $y^{pbl}$ within local scope
end Identify-last-definition
function Identify-parent-declarations
     move left & up to mark parent of $X^{pbl}$
  if $X^{pbl}$ is not used in the last definition, move left & up to mark declarations
end Identify-parent-declarations
```

Figure 17: General static program slicing algorithm

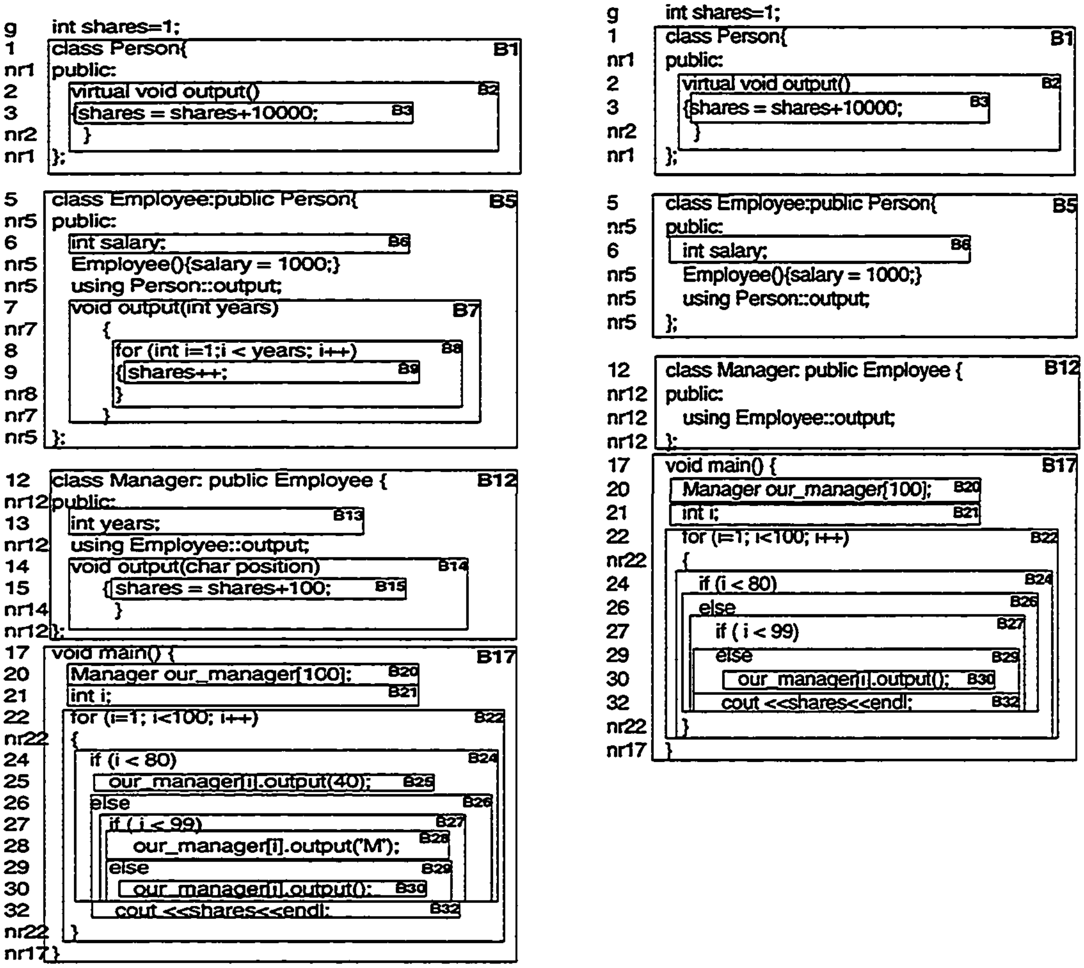
The static program slicing algorithm utilizes two methods for navigating through the syntax tree. The first navigation method *"left & up"* starts left from the block of interest and then moves up towards the root block. The second navigation method "down & right" identifies blocks containing the last definitions of variables within the given scope. The navigation method is responsible for positioning the pointer at the current analysis position and identifying the last block that is connected syntactically to the starting block. The navigation algorithms are exemplified in Figure 15 using bold arrows. The starting point, by default, is from the last block of the syntax tree if there is no specific block requested. The navigation is initiated with a *"left & up"* method, by default, from the user-selected starting point on the syntax tree and the starting point is "marked" in the beginning of the algorithm.

In the example, we start from the last block (statement 32) with variable of interest "*shares*", which is automatically "marked". Both navigation methods are used to find the previous definition of the current variable(s) by traversing recursively through the syntax tree. The navigation path for a complex block is illustrated in Figure 16 and is self-explanatory. To derive a static program slice for a pair (V,n) containing a set of variables V and starting point n, we have to traverse through all the relevant blocks of the syntax tree. . In the example, navigation for, $V =$ *"shares"* and n =32 is shown. The algorithm has two separate methods for marking and visiting, and they use the navigation methods "*left & up*", and "*down & right*" as mentioned above. The slicing algorithm starts by searching for the last "marked" (or highest block number of the particular tree) from the current block using "left & up".

Whenever the navigation encounters a "*function*" call in a block tree, it checks for re-definition of the current variable either in the form of pointer reassignment or as a global variable for later processing. The navigation will then continue until it finds a new definition within the current syntax tree or it encounters a root block of the syntax tree. If the "marked" block belongs to a complex block like "if-else, while, for, etc." then further navigation is carried out until the beginning of that complex block without terminating the search. All the possible blocks that define, the current variable within the complex block scope are "marked".

Once the last definition for all used variables at the current block are analyzed, then the current block will be marked as "visited" indicating that it is not a removable block. If the current block has a function call, then the navigation on the current syntax tree is suspended to traverse through the new syntax tree that includes function definition. In the algorithm, "*goto*" and "*label*" blocks are considered contributing blocks if the particular outer most complex block encompassing both of the constructs is a non-removable block. If the above condition is satisfied then they are considered non-removable blocks. They are "marked" and "visited" during the traversing of the syntax tree along with others. The algorithms also consider the outer blocks until the root block as their contributing blocks and they are therefore not removable.

Any derived object that is included in a slice will have its parent class constructs included in the slice along with its own constructs including all the class variables. However, user defined methods in the parent class are not part of the slice unless a derived object method uses a specific parent's method. Figure 18 shows the extracted slices from the new static and Korel's [30] dynamic slicing algorithms.



```
g     int shares=1;
1     class Person{
nr1   public:
2       virtual void output()
3       {shares = shares+10000;
nr2      }
nr1   };

5     class Employee:public Person{
nr5   public:
6       int salary;
nr5     Employee(){salary = 1000;}
nr5     using Person::output;
7       void output(int years)
nr7        {
8           for (int i=1;i < years; i++)
9           {shares++;
nr8         }
nr7        }
nr5   };

12    class Manager: public Employee {
nr12  public:
13      int years;
nr12    using Employee::output;
14      void output(char position)
15         {shares = shares+100;
nr14        }
nr12  };
17    void main() {
20      Manager our_manager[100];
21      int i;
22      for (i=1; i<100; i++)
nr22    {
24        if (i < 80)
25          our_manager[i].output(40);
26        else
27          if ( i < 99)
28            our_manager[i].output('M');
29          else
30            our_manager[i].output();
32          cout <<shares<<endl;
nr22    }
nr17  }
```

```
g     int shares=1;
1     class Person{
nr1   public:
2       virtual void output()
3       {shares = shares+10000;
nr2      }
nr1   };

5     class Employee:public Person{
nr5   public:
6       int salary;
nr5     Employee(){salary = 1000;}
nr5     using Person::output;
nr5   };

12    class Manager: public Employee {
nr12  public:
nr12    using Employee::output;
nr12  };
17    void main() {
20      Manager our_manager[100];
21      int i;
22      for (i=1; i<100; i++)
nr22    {
24        if (i < 80)
26        else
27          if ( i < 99)
29          else
30            our_manager[i].output();
32          cout <<shares<<endl;
nr22    }
nr17  }
```

Figure 18: Static and dynamic program slice for variable *"shares"* at statement 32

3.3 Extended version of the static slicing algorithm

With the extended algorithm, an attempt is made to improve the accuracy of algorithm with respect to the handling of polymorphism and therefore allowing for the computation of smaller slices. In an object-oriented program, a polymorphic reference can, over time, refer to instances of more than one class. Therefore, the static representation should represent this dynamic feature of the object-oriented paradigm. In C++, polymorphic method calls occur with an indirect de-reference, and the type of the referenced object is unknown at the time [59]. In the presented general static program slicing algorithm, a conservative approach is used for polymorphism to overcome this problem and it allows including all the methods in the related group of classes. For the extended version of static program slicing algorithm, optimized handling of polymorphic function call is introduced. Optimization is carried out by checking and matching related variable types in the method calls with the method definition of groups of related classes to identify a particular function call. If extended algorithm identifies a non-resolvable match, then it will apply previous conservative approach as in original algorithm and will include all the possible candidates in the slice.

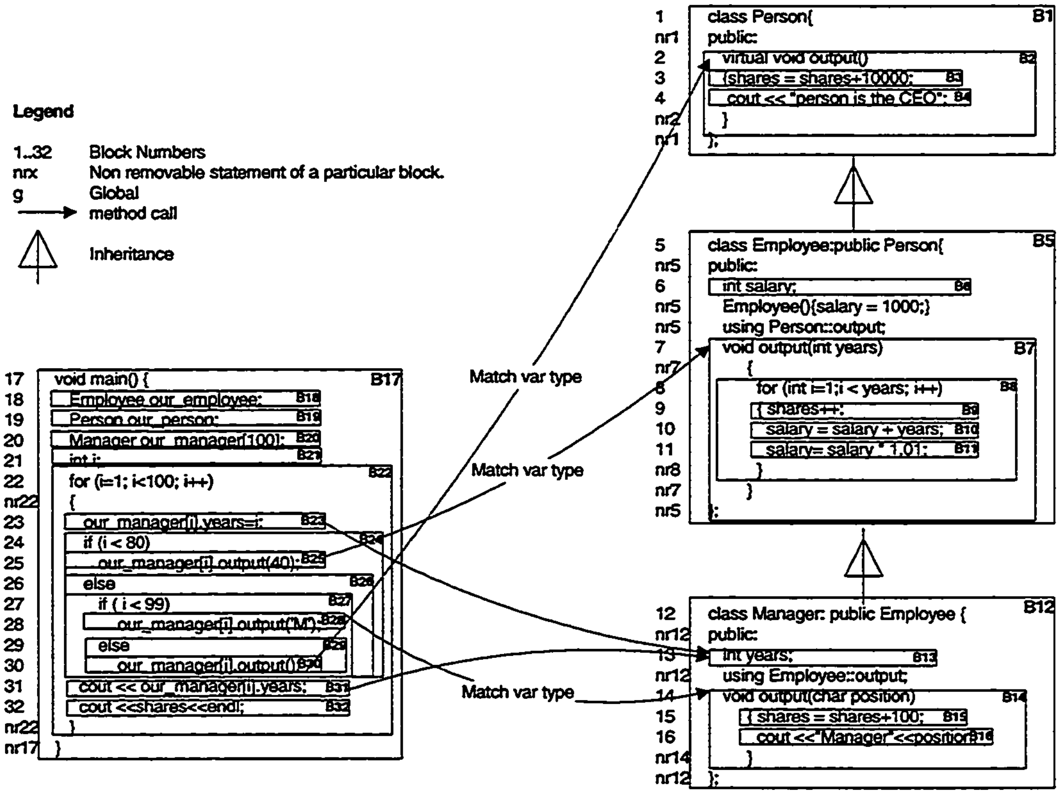


Figure 19: Function prototype matching

The presented static program slicing algorithm and criterion based hybrid slicing algorithm share a common characteristic: the notion of removable blocks. Another additional concept that is common among the algorithms is $N(B)$ that represents a set of all nodes (statements) that belong to block B.

3.3.1 Proof of correctness of generic static program slicing algorithm

The same theorems and lemmas presented in [30] to prove the correctness of the algorithm also applies to the following generic static program slicing algorithm. The following section provides a summary of the theorems and lemmas.

Theorem:

The generic static program slicing algorithm presented in Figure 17 correctly computes a static program slice for the slicing criterion $C=(y^{sbl})$.

In order for the generic static program slicing algorithm to compute correct static slices, the algorithm has to satisfy Lemma 1 and 2 presented in [30]. The same formal proof might be applied for the general static program slicing algorithm. $N(B)$ means a set of all statements (r-blocks) that belong to block B. Let $T_{Trace}(B,pbl_1,pbl_2)$ be a tree trace in contrast block trace in [30] where pbl_1 and pbl_2 represents *r-entry* and *r-exit* respectively. $M(T_{Trace})$ means a set of all blocks in a given tree trace Tr_{bl}. i.e.,

$M(Tr_{bl}) = \{y^{pbl} \mid Tr_{bl}(pbl) = y$ and $1 \leq pbl \leq sbl\}$. Two tree traces $T_{Trace}(B_1,pbl_1,pbl_2)$ and $T_{Trace}(B_2,tbl_1,tbl_2)$ are disjoint if they do not contain the same block.

We create the following loop invariant, presented at line#8 of the algorithm in Figure 17.

1. D_{bl} is a set of all blocks set as marked or visited in Tr_{bl}.

$$M(Tr_{bl}) - \bigcup M(T_{Trace}(B,pbl_1,pbl_2)) = D_{bl}, \text{ where } T_{Trace}(B,pbl_1,pbl_2) \in \Phi_{bl}$$

2. $LD(y^{sbl}) \in I_{bl}$,

3. for all blocks $X^{pbl} \in I_{bl}$ variables $v \in U(X^{pbl})$, $LD(v^{pbl}) \in I_{bl}$,

2. for all blocks $X^{pbl} \in I_{bl}$, for all blocks $B \in R_{bl}$, $X \notin N(B)$.

Proof of the loop invariant

The loop invariant at line 8 is true for each of the loop iterations. Notice that I_{bl} in the algorithm is a set of all blocks that are set as visited, whereas D_{bl} is a set of all blocks that are set as marked or visited. Therefore, I_{bl} is a subset of D_{bl}. Steps 7 and 13(second part *Identify parents and declarations,* which are neutral in some cases) satisfy condition 1 of the loop invariant. Step 7 ensures that all the identified tree traces that are included in Φ_{bl}, exclude blocks set as visited (lines 27, 28 and 34). Notice that all tree traces in Φ_{bl} are disjoint because when a tree trace $T_{Trace}(B,pbl_1,pbl_2)$ is found (in lines 30-35), the search continues from the first block following the last block in $T_{Trace}(B,pbl_1,pbl_2)$ In addition, only traces of tree that belong to R_{bl} are considered (line 30). Condition 2 of the loop invariant is true because step 4 marks the last definition of y^{sbl}. Condition 3 is satisfied because for each block X^{pbl} that is set as visited in step 15 all last definitions of all variables used in X^{pbl} are set as marked in step 15&16 (if they were not marked before). All these last definitions are visited and inserted in I_{bl} on the next iterations of the while-loop 11-21. Condition 4 is satisfied because for each block X^{pbl} that is set as visited in step 13, all blocks to which X belongs is removed from R_{bl} in step 19.

Proof of correctness of the theorem

A proof of the correctness of the theorem on the generic static program slicing algorithm is presented Figure 17 computes correct static program slices for the slicing criterion $C=(y^{sbl})$.

On the termination of the repeat loop 5-8 of the algorithm, the loop invariant is combined with the termination condition. Since on the termination of the loop there are no more marked and not-visited blocks in the tree trace, D_{bl} equals to I_{bl} ($D_{bl} = I_{bl}$). Combining this condition with the loop invariant, the following conditions are true on the exit from the repeat loop:

1. $M(Tr_{bl})- \bigcup M(T_{Trace}(B,pbl_1,pbl_2)) = I_{bl}$ where $T_{Trace}(B,pbl_1,pbl_2) \in \Phi_{bl}$
2. $LD(y^{sbl}) \in I_{bl}$,
3. for all blocks $X^{pbl} \in I_{bl}$ variables $v \in U(X^{pbl})$, $LD(v^{pbl}) \in I_{bl}$,
4. for all blocks $X^{pbl} \in I_{bl}$, for all blocks $B \in R_{bl}$, $X \notin N(B)$.

As a result, a static program slice constructed from program P by removing all blocks that belong to R_{bl} from it, is a static program slice for the slicing criterion $C=(y^{sbl})$ and is an correct executable slice.

3.3.2 Application of general static program slicing algorithm

The static program slicing algorithm (Figure 17) is applied on the sample program shown in Figure 13. Figure 20 presents the algorithm iteration details. The step 13 is used in last column which marks the statement based either parent or based on inheritance. Φ_{bl} column shows the block that are non-contributing blocks as identified by the algorithm. As one can see that at the end, blocks in R_{bl} is same as Φ_{bl} and are non-contributing. By removing these blocks, behavior of the program will not change for the variable *"shares"* at statement 32. The step 13 repeatedly marks the same parent or inherited node such that it will make sure that all the inheritance and parent blocks are included to preserve the executable property of program slice.

	I_{bl}	R_{bl}	Φ_{bl}	Marked by line 13
1	{32}	{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31}	{31, 24,31}	22,21,17
2	{32,30}	{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,31}	{4, 3,4}	29,27,26,24,22,21,17
3	{32,30,3}	{1,2,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,31}		2,1
4	{32,30,3,2,1}	{4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,31}		29,27,26,24,22,21,17
5	{32,30,3,2,1,29}	{4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,31}		27,26,24,22,21,17
6	{32,30,3,2,1,29,28}	{4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,31}		27,26,24,17
7	{32,30,3,2,1,29,28,15}	{4,5,6,7,8,9,10,11,12,13,14,16,17,18,19,20,21,22,23,24,25,26,27,31}	{16, 15,16}{13, 12,13}	14,12,5,1
8	{32,30,3,2,1,29,28,15,14,12}	{4,5,6,7,8,9,10,11,13,16,17,18,19,20,21,22,23,24,25,26,27,31}		5,1
9	{32,30,3,2,1,29,28,15,14,12,27,26}	{4,5,6,7,8,9,10,11,13,16,17,18,19,20,21,22,23,24,25,31}		24,22,17
10	{32,30,3,2,1,29,28,15,14,12,27,26,25}	{4,5,6,7,8,9,10,11,13,16,17,18,19,20,21,22,23,24,31}	{10, 9,10}{11, 10,11}	24,22,17
11	{32,30,3,2,1,29,28,15,14,12,27,26,25,9}	{4,5,6,7,8,10,11,13,16,17,18,19,20,21,22,23,24,31}		8,7,5,1
12	{32,30,3,2,1,29,28,15,14,12,27,26,25,9,8}	{4,5,6,7,10,11,13,16,17,18,19,20,21,22,23,24,31}		7,5,1
13	{32,30,3,2,1,29,28,15,14,12,27,26,25,9,8,7,6,5}	{4,10,11,13,16,17,18,19,20,21,22,23,24,31}		1
14	{32,30,3,2,1,29,28,15,14,12,27,26,25,9,8,7,5,6, 24}	{4,10,11,13,16,17,18,19,20,21,22,23,31}	{23,22,23}	22,17
15	{32,30,3,2,1,29,26,15,14,12,27,26,25,9,8,7,5,24,22}	{4,10,11,13,16,17,18,19,20,21,23,31}		17
16	{32,30,3,2,1,29,28,15,14,12,27,26,25,9,8,7,5,6,24,22,21}	{4,10,11,13,16,17,18,19,20,23,31}		17
17	{32,30,3,2,1,29,28,15,14,12,27,26,25,9,8,7,5,6,24,22,21,20}	{4,10,11,13,16,17,18,19,23,31}	{19,18,19}{18,17,18}	17
18	{32,30,3,2,1,29,28,15,14,12,27,26,25,9,8,7,5,6,24,22,21,20,17}	{4,10,11,13,16,18,19,23,31}		17

Figure 20: Application of general static slicing algorithm

(P,v) = (Sample program shown in Figure 13, *"shares"* at statement 32)

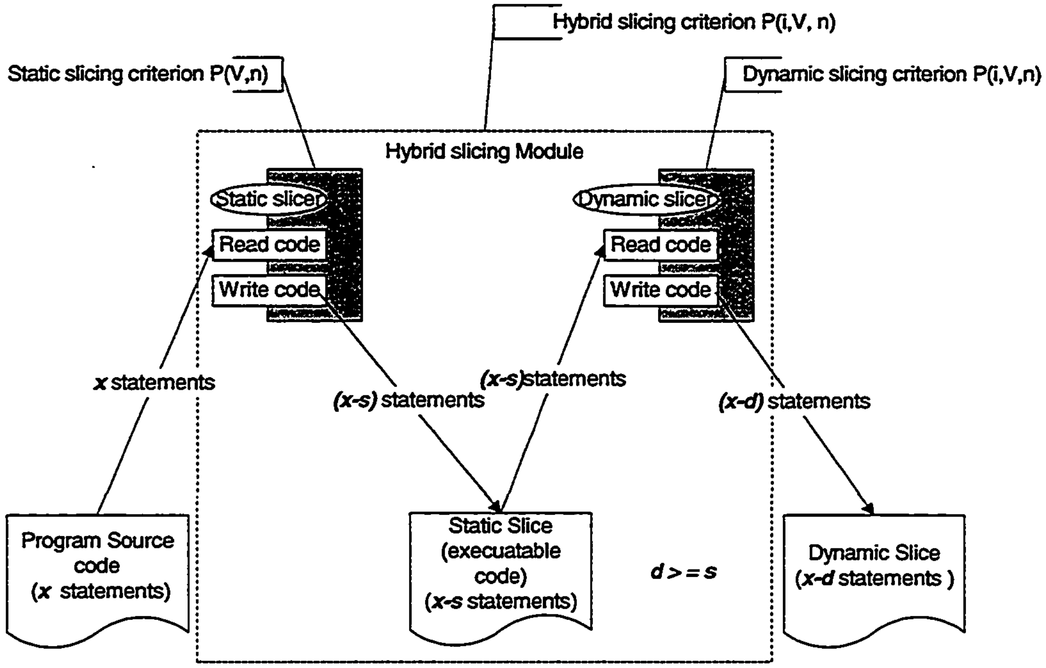
4. Enhanced hybrid program slicing based on removable blocks

The hybrid program slicing techniques are introduced to use the best of both static and dynamic program slicing. In particular it uses the accuracy in handling dynamic language constructs from dynamic program slicing. However, it uses static program slicing to reduce the computation of dynamic slices (for example see Figure 21) that incurs a high run time overhead due to recording the program execution and/or analysis of every executed statement. The time and space required for recording execution trace and traversing for analyzing is reduced either supplementing the original code with a statically reduced code or applying static slicing criterion within the dynamic slicing. In the next section, two hybrid program slicing algorithms are introduced that represent the part of program slicing framework incorporated in MOOSE. The two hybrid program slicing algorithms combine the properties of the general static and dynamic program slicing algorithms, and they utilize common information between the two algorithms to enhance their usability for a particular task.

4.1 Basic hybrid program slicing algorithm

The first hybrid program slicing algorithm is conceptually a combination of a pre-processed static program slice that provides the input (based on a specific slicing criterion *P(V,n)*) to a dynamic program slicing algorithm (for the refined slicing criterion *P(i, V, n)* with input *i*). The static program slice conservatively includes all the possible polymorphic routes that are not uniquely identifiable by the algorithm. The approach for the basic hybrid program slice algorithm is as follows: (1) First, a static program slice for a slicing criterion *P(V,n)* is computed. This static program slice reduces the program size by eliminating all statements from the original program that are not relevant for the computation of the slicing criterion without changing the program's behavior. In other words, it will help to project the statements that are used for the slicing criterion. (2) The static program slice then will become the input for the dynamic slicing algorithm computing a dynamic program slice for the refined slicing criterion *P(i,V,n),* but for a particular program execution rather than all possible program executions. Figure 21 illustrates the concept of basic hybrid program slicing.

Utilizing the reduced static program slice as input to the dynamic slicing algorithm rather than the original source code allows a reduction in the number of statement that will have to be executed and analyzed by the dynamic slicing algorithm. Only at the first time for a particular slicing criterion *P(V,n)*, both the static and dynamic program slice, have to be generated. Later, only dynamic program slicing algorithm needs to be revisited for evaluating the boundary behavior of the slice variety of inputs within the slicing criterion *P(V,n).*



Legend:

P(V,n): Set of variables *V* at statement *n*.
P(i,V,n): Set of variables *V* with input value *i* at statement *n*
x :Number of statements in source code
s: Number of statements removed in static slice
d: Number of statements removed in dynamic slice.

Figure 21: Concept of basic hybrid program slicing algorithm

The major advantages of this algorithm are as follows: (1) Shorter program executions, because the static program slice contains fewer statements that have to be executed and analyzed by the dynamic program slicing algorithm. (2) There is no loss in accuracy. The algorithm will still provide the accuracy of the regular dynamic program slice. (3) The computation of the basic hybrid algorithm can be made completely transparent to the user.

The main disadvantage of this algorithm is that it will only provide significant improvements if the static slicing algorithm can provide a significant reduction in the source code size and the number of executed statements for the slicing criterion *P(V,n)*.

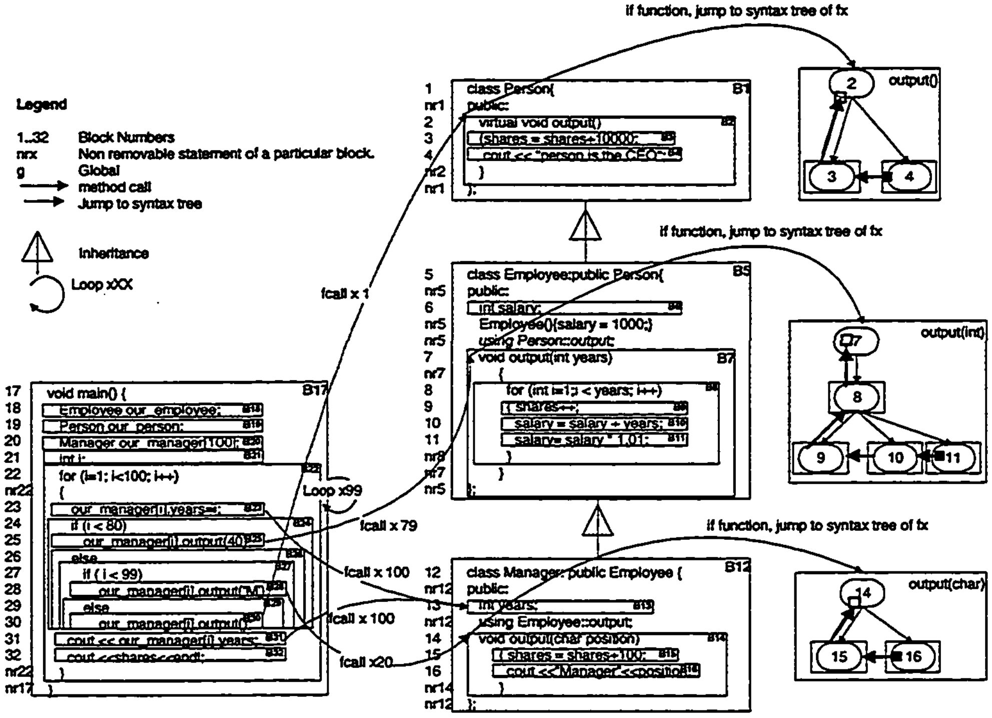
4.2 Criterion based hybrid program slicing algorithm.

The criterion based hybrid program slicing algorithm is an enhanced dynamic program slicing algorithm that provides options to users to use static slicing algorithm for certain language constructs. The static program slicing information is used to reduce the overhead involved in recording and analyzing all the executed statements. If the user selects no static criterion, then it will compute pure dynamic program slice, on contrary if the user selects the entire available static criterion, then it will compute more or less static program slice.

The criterion based hybrid program slicing algorithm takes the advantages of both the general static and dynamic program slicing algorithms are based on removable blocks. The criterion based hybrid program slicing algorithm involves three different steps: (1) The user selects a slicing criterion *P(V,n),* complete syntax tree for *P* is created and stored. (2) The user selects certain language constructs, such as function calls, loops etc, that should be handled by the static program slicing algorithm utilizing static information, rather than analyzing these statements dynamic program slicing algorithm. (3) Program's execution trace T_x is recorded excluding the user selected language constructs.

Whenever, the hybrid program slicing algorithm encounters one of the selected languages constructs during program analyses, the algorithm will suspend its analysis and switch to the static program slicing algorithm. After completing the static analysis, the algorithm will continue with hybrid program slicing.

The static program slicing algorithm will determine the contribution of each block within the selected construct and its associated blocks. An example is shown in Figure 22 where the user has selected all the procedure calls that shall be avoided for dynamic program slicing, instead static program slicing is used on these procedures.



```
17   void main() {
18   Employee our_employee;
19   Person our_person;
20   Manager our_manager[100];
21   int i;
22   for (i=1; i<100; i++)
nr22 {
23   our_manager[i].years++;
24   if (i < 80)
25   our_manager[i].output(40);
26   else
27   if ( i < 99)
28   our_manager[i].output("M");
29   else
30   our_manager[i].output();
31   cout << our_manager[i].years;
32   cout <<shares<<endl;
nr22 }
nr17 }
```

```
1    class Person{
nr1  public:
2    virtual void output()
3    {shares = shares+10000;
4    cout << "person is the CEO";
nr2  }
nr1  };
```

```
5    class Employee:public Person{
nr5  public:
6    int salary;
nr5  Employee(){salary = 1000;}
nr5  using Person::output;
7    void output(int years)
nr7  {
8    for (int i=1;i < years; i++)
9    { shares++;
10   salary = salary + years;
11   salary= salary * 1.01;
nr8  }
nr7  }
nr5  };
```

```
12   class Manager: public Employee {
nr12 public:
13   int years;
nr12 using Employee::output;
14   void output(char position)
15   { shares = shares+100;
16   cout <<"Manager"<<position;
nr14 }
nr12 };
```

Figure 22: Criterion based hybrid program slicing

In the above example, every time the execution encounters *our_manager(I).ouput(40),* the program identifies it for static program slicing of function *output(int years)* and at the same time, the program suspends the recording of the execution trace until it returns. For the example above, the hybrid algorithm will reduce the required recording/dynamic analysis of the execution trace for the function *output (int years)* by 79 x 40 statements. Clearly, the hybrid program algorithm reduces the space and time complexity of the slice computation. The final hybrid slice is derived by combining the partial dynamic program slice with the partial static program slice. The accuracy of the hybrid program slice will be higher or equal to the static program slice, but less than or equal that of the dynamic program slice, depending on the number and type of criterions selected to handle statically.

4.2.1 Description of Algorithm

The presented algorithm (Figure 23) uses the definitions stated in static and dynamic program slicing algorithms discussed in the previous section. Algorithm identifies a set of removable blocks R_C. In line #1 of the algorithm, a syntax tree Tr_{bl} for program P is initiated. In line#2 of the algorithm, the program is executed and its execution trace is recorded up to the execution position q excluding *LaBlock* that is to be handled statically. In line#3, set R_C is initialized with all blocks of program P. In line#4, all actions in the execution trace are set as unmarked and not visited . In addition, blocks of Tr_{bl} are also set as unmarked and not visited. In line#5, a set of contributing blocks $I_c = I_{bl} = \emptyset$, is initialized as an empty set along with R_{bl} which will later hold the blocks based on *LaBlock*. In line#6 the algorithm identifies and marks the last definition y^p of variable y^q, more details of this procedure is presented in line# 97- 101. The algorithm iterates in the repeat loop in line# 7-15 until all marked actions are visited. There are five major steps inside of the repeat loop. In line# 8, the algorithm checks if a contributing action (C_c) is to be handled statically (*LaBLock*).

Algorithm

Input: a slicing criterion $C=(x, y^q)$

Output: a hybrid program slice for C

Tr_{bl}: Tree with bl blocks

T_x execution trace up to position q

Φ_c: Set of block traces

R_c: Set of blocks (dynamic)

$R_{c\,(LaBlock)}$: Set of blocks (dynamic) used by the selected criterion.

$Tr_{bl(\,LaBlock\,)} \in Tr_{bl}$, Tree trace relevant to $R_{c\,(\,LaBlock)}$

I_c: set of contributing actions (default is visited)

N_c non-contributing action(dynamic)

R_{bl}: Set of blocks (static)

I_{bl}: Set of contributing blocks (default is visited)

N_{bl} non-contributing blocks (static)

C_{bl} contributing blocks (static)

x^{bl} current evaluated block position

$y^q = y^{sbl}$ Variable y at simple block s where $s \leq q$

```
Create static $Tr_{bl}$ for $P$
Execute program $P$ on input $x$ and record execution trace $T_x$ up to position $q$, do
not record $LaBlock$
Initialize $R_c$ to a set of all blocks in program $P$
Set all actions in $T_x$ and all blocks in $Tr_{bl}$ as not marked and not visited.
$I_c = I_{bl} = R_{bl} = \emptyset$
Find last definition $y^p$ of $y^q$ and set $y^p$ as marked
repeat
    If $C_c = LaBlock$
                $R_{bl} = R_{bl} \cup \{R_{c(\,LaBlock)}\}$
                $R_c = R_c - \{\,R_{c\,(\,LaBlock)}\}$
                Identify $Static\_analysis$ for $LaBlock$
        Else    Identify $C_c\_Dynamic$
                Identify $N_c\_Dynamic$
    Mark remaining actions
until every marked action in $T_x$ was visited
Create Hybrid program slice by removing all blocks $\in (\,R_c \cup R_{bl})$
```

Figure 23: Criterion based hybrid-program slicing algorithm (continued)

```
function Static_analysis
Find last definition $y^{pbl}$ of variable $y^{sbl}$ and set $y^{pbl}$ as marked
repeat
        Identify $C_{SG}$_Static
        Identify $N_{SG}$_ Static
until every marked block in $Tr_{bl(LaBlock)}$ is visited
end Static_analysis
function Identify $C_{SG}$_Static
while(contributing + not visited statement in LaBlock)
Select a contributing and not visited block $X^{bl}$ in $Tr_{bl(LaBlock)}$
 Set $X^{bl}$ as a visited block & Identify-parent-declarations, Identify-inheritance
 $I_{bl} = I_{bl} \cup \{ X^{bl} \}$
for all variables $v \in U(X^{bl})$ do
  Identify + mark last definition of $v$ as contributing block
    end-for
for all blocks $B_{LaBlock} \in R_{bl}$ do
     if $X^{bl} \in N(B_{LaBlock})$ then $R_{bl} = R_{bl} - \{ B_{LaBlock} \}$
end-for
end-while
end Identify $C_{SG}$_Static
function Identify $C_C$
 while (contributing and not visited action in $T_X$)
    Select a contributing and not visited action $X^k$ in $T_X$
     Set $X^k$ as a visited action
    $I_C = I_C \cup \{X^k\}$
    for all variables $v \in U(X^k)$ do
       Identify + mark last definition of $v$ as contributing action
    end-for
    for all blocks $B \in R_c$ do
      if $X \in N(B)$ then $R_c = R_c - \{B\}$
 end-while
 end Identify $C_C$
function Identify $N_{SG}$_Static
$\Phi_{bl} = \emptyset$
pbl = last marked and visited block
do
        Let $X^{pbl}$ be a block at position pbl in $Tr_{bl}$
```

Figure 23: Criterion based hybrid-program slicing algorithm (continued)

```
if $X^{pbl} \notin I_{bl}$ then
  $y^{tbl} \in I_{bl}$ be the closest $C_{bl}$ from position $pbl$ by navigating
          the syntax flow (left & up / down & right) such that $tbl<pbl$
  if there exists block $B_{LaBlock} \in R_{bl}$ such that
          (1) $B_{LaBlock}$ has an r-exit at position $pbl$, and
          (2) there exists no inheritance and function call
          (3) $B_{LaBlock}$ has an r-entry between $tbl$ and $pbl$ then
      Let $pbl_1$ be the closest position of an r-entry of $B_{LaBlock}$ from position $pbl$, where $tbl<pbl_1<pbl$
          $\Phi_{bl} = \Phi_{bl} \cup \{ T_{Trace} (B_{LaBlock}, pbl_1, pbl) \})$
          $pbl_1 = pbl - 1$ by navigating the syntax flow (left & up / down & right)
      end-if
  end-if
  $pbl := pbl - 1$
 while(contributing + not visited statement in LaBlock)
End- Identify $N_{SG\_}$Static
function Identify $N_C$
$\Phi_C = \emptyset$
$p = 1$
do
   Let $X^p$ be an action at position $p$ in $T_x$
   if $X^p \notin I_C$ then
     $Y^t \in I_C$ the closest $C_C$ from position $p$ such that $t>p$
      if there exists block $B \in R_c$ such that
        (1) $B$ has an r-entry at position p, and
        (2) there exists no inheritance
        (3) B has an r-exit between $p$ and $t$ then
           $p_1$ is closest r-exit of B from position $p$, where $p<p_1<t$
            $\Phi_C = \Phi_C \cup \{S(B,p,p_1)\})$
            $p = p_1 - 1$
      end-if
  end-if
  $p := p + 1$
while $p \geq q$
end Identify $N_C$
function Mark remaining actions
for all actions $X^k$ that are not identified $C_C$ nor as $N_C$
    Set $X^k$ as marked
end-for
```

Figure 23: Criterion based hybrid-program slicing algorithm (continued)

```
91.  end Mark remaining actions
92.  function Identify inheritance
93.  if current class is derived then
94.     set all $X^{base}$ classes as contributing
95.  end-if
96.  end Identify-inheritance
97.  function Find last definition
98.    Find last definition of variable $y^p$ in global scope
99.    If not found last definition then
100.       search last definition of $y^p$ within local scope
101. end Identify-last-definition
102. function Identify-parent-declarations
103.     move left & up to mark parent of $X^{pbl}$
104.     if $X^{pbl}$ is not used in the last definition, move left & up to mark declarations
105. end Identify-parent-declarations
```

Figure 23: Criterion based hybrid-program slicing algorithm

If the language construct is to be handled statically, the algorithm removes set of blocks $Rc_{(LaBlock)}$ belong to the particular *LaBLock* and transfers it to the R_{bl} in line#9 and 10. Later in line# 11, call for *Static_analysis* made to analyze the blocks that contributes to the computation of $y^q (=y^{sbl})$ in syntax tree. In line# 17-23. *Static_analysis* procedure is shown. *IdentifyC$_c$_Dynamic* in line#12 the algorithm identifies the actions that contribute to the computation of y^q using the execution block trace. Similarly, based on the outcome of line# 8, line#13 identifies, non-contributing actions by finding a set of block traces. In line#14, all the remaining actions/statements, i.e., actions that are not identified either contributing or non-contributing actions, are marked. These actions are visited on the next iteration of the repeat loop. The process of identifying contributing and non-contributing actions continues in the repeat loop until all actions are classified as either contributing or non-contributing actions. The hybrid program slice is derived by removing the blocks belonging to $(R_c \cup R_{bl})$, in other words by combining both dynamic and static non-contributing set of blocks.

Static_analysis procedure (line# 17-23) uses part of the tree trace Tr_{bl} dictated by the *LaBLock*. Similar to static program slicing algorithm, this procedure iterates in the repeat loop line# 19-23 until all marked blocks are visited within the scope of *LaBLock*. In line# 20, the algorithm identifies blocks that contribute to the computation of y^{sbl}. In line# 21, for the given set of contributing blocks, the algorithm identifies non-contributing blocks by finding a set of tree traces. It is important to mention that, for the hybrid program slicing, no block gets both dynamic and static analysis. In other words, only one type of analysis applied for each of the block.

All other procedures are similar to either static or dynamic program slicing and are explained along with the algorithms presented in their respective sections. For example, "function *Identify* C_{SG}*_Static*" hybrid program slicing is similar to "function *Identify* C_{bl}" of static program slicing and so are the others, hence they are not explained in detail here to avoid the repetition. The major difference between them is the scope of analysis, in hybrid program slicing, only selected part of the whole tree trace is analyzed where as in static program slicing the whole tree trace is analyzed.

The major advantages of this algorithm are (1) it provides the user with the ability to select the properties that are most important for him/her for the specific task (either accuracy of the slice or time and space complexity). (2) The user can choose to have all language constructs such as function calls, different loops being handled statically to reduce the time and space complexity required by the hybrid program slicing algorithm.

The disadvantages of this algorithm are (1) it compromises the accuracy to maximize the time and space savings as compared to the pure dynamic program slice computation. (2) In some cases, expected time and space savings from the static analysis are not significantly different from dynamic analysis, the dual computation of static and dynamic slices might cause extra overhead.

4.2.2 Proof of correctness of hybrid program slicing algorithms

As stated in previous section, the proofs of lemmas presented in [30] to prove the correctness of the algorithm also apply to the following hybrid program slicing algorithms. For the "Basic hybrid program slicing algorithm" (BHPSA), proof is straight forward as it uses both proven algorithms in sequence. First, it uses static program slicing algorithm (SPSA) to get static slice. Later, it uses proven dynamic program slicing algorithm (DPSA) get the final slice. As both algorithms proved true in different occasions [30], their union is also true. Mathematically,

SPSA =True, DPSA = True (Proved in previous section and in [30])

(SPSA Λ DPSA) $\Rightarrow$ BHPSA.

True Λ True $\Rightarrow$ BHPSA

BHPSA $\Rightarrow$ True

The following section provides a summary of the theorems and lemmas for criterion based hybrid-slicing algorithm.

Theorem:

The criterion based hybrid program slicing algorithm presented in Figure 23 that correctly computes a hybrid program slice for the slicing criterion $C=(x, y^q)$.

In order to compute correct hybrid program slices, the criterion based hybrid program slicing algorithm has to satisfy Lemma 1 and 2 presented in [30]. The same formal proof might be applied for the hybrid program slicing algorithm.

4.2.3 Definitions used in static and dynamic program slicing algorithms

The following definitions are repeated from static program slicing and dynamic program slicing for ready reference.

Static program slicing algorithm:

Let $T_{Trace}(B,pbl_1,pbl_2)$ be a tree trace in contrast block trace in [30] where pbl_1 and pbl_2 represents *r-entry* and *r-exit* respectively. $M(T_{Trace})$ means a set of all blocks in a given tree trace Tr_{bl}.. i.e.,

$M(Tr_{bl}) = \{y^{pbl} \mid Tr_{bl}(pbl) = y$ and $1 \leq pbl \leq sbl\}$. Two tree traces $T_{Trace}(B_1,pbl_1,pbl_2)$ and $\mathrm{T}_{\mathrm{Trace}}(B_2,tbl_1,tbl_2)$ are disjoint if they do not contain the same block.

Dynamic program slicing algorithm:

Let $S(B, k_1, k_2)$ be a block trace; by $M(S(B, k_1, k_2))$ we denote a set of all actions that belong to $S(B, kl, k2)$. Similarly, $M(T_X)$ denotes a set of all actions in a given execution trace T_x, i.e., $M(T_x)=\{Y^p \mid T_x(p)=Y \text{ and } 1 \leq p \leq q\}$. Two block traces $S(B_1,k_1,k_2)$ and $S(B_2,t_1,t_2)$ are *disjoint* if they do not contain the same action.

4.2.4 Definitions of criterion based hybrid program slicing algorithm

The following are defined for criterion based hybrid program slicing algorithm:

Set of all blocks of program P is unique for both static and dynamic program slicing, in all cases $S(B_s,k_1,k_2) \neq T_{Trace}(B_t,pbl_1,pbl_2)$ as $pbl_1 \& pbl_2 \neq k_1 \& k_2$

Also, for unique slice, $R_{bl} \cap R_c = \emptyset$ and $I_{bl} \cap I_c = \emptyset$ and $\Phi_{bl} \cap \Phi_c = \emptyset$

Let R_{hc} be the removable blocks for correct slice of the hybrid slicing, then

$$R_{hc} = R_{bl} \cup R_c$$

The criterion based actions in a given execution trace $T_x = M_C(T_x)$ and similarly criterion based blocks used in criterion base tree trace $T_{Trace} = M_{CTrace}(T_{Trace})$

The following loop invariant is created from line 15 and 22 of the algorithm in Figure 23.

1. D_C is a set of all actions set as marked or visited in T_x and D_{bl} is a set of all blocks set as marked or visited in Tr_{bl} and $D_C \cap D_{bl} = \emptyset$

$$M_C(T_x) - \bigcup_{S(B,k_1,k_2) \in \Phi_C} M_C(S(B,k_1,k_2)) = D_C,\; M_{laBlock}(Tr_{bl}) - \bigcup_{T_{Trace}(B,pbl_1,pbl_2) \in \Phi_{bl}} M_{CTrace}(T_{Trace}(B_{bl},pbl_1,pbl_2)) = D_{bl}$$

and $\Phi_{bl} \cap \Phi_c = \emptyset$

2. $LD(y^q) \in I_c$ and $LD(y^{sbl}) \in I_{bl}$ where $I_{bl} \cap I_c = \emptyset$
3. for all actions $X^k \in I_c$ variables $v \in U(X^k)$, $LD(v^k) \in I_c$ and for all blocks $X^{pbl} \in I_{bl}$ variables $v \in U(X^{pbl})$, $LD(v^{pbl}) \in I_{bl}$ and $I_{bl} \cap I_c = \emptyset$
4. for all actions $X^k \in I_c$, for all blocks $B \in R_c$, $X \notin N(B)$ and, for all blocks $X^{pbl} \in I_{bl}$, for all blocks $B_{bl} \in R_{bl}$, $X \notin N(B)$ and $R_{bl} \cap R_c = \emptyset$

Proof of the loop invariant

The loop invariant at line 15 and 22 is true for each of the loop iterations. Notice that I_c and I_{bl} in the algorithm are a set of all blocks that are set as visited. D_C and D_{bl} are set of all blocks corresponding to that I_c and I_{bl} that are set as marked or visited. Therefore, I_{bl} is a subset of D_{bl} similarly I_c is a subset of D_C. Steps 12, 20, 40 and 27 (second part Identifies parents and declarations, which are neutral in some cases) satisfy condition 1 of the loop invariant. Step 13 and 21 ensures that all the identified tree traces that are included in Φ_c (72,73 and 79) and Φ_{bl}, (lines 53,54,60) exclude blocks set as visited Notice that all tree traces in Φ_{bl} are disjoint. This is because tree trace $T_{Trace}(B,pbl_1,pbl_2)$ is found (in lines 56-61), the search continues from the first block following the last block in $T_{Trace}(B,pbl_1,pbl_2)$. In addition, only traces of tree that belong to R_{bl} are considered (line 56). Similarly all block traces for Φ_c are disjoint and the block trace $S(B,p,p_l)$ is found,(lines 75-80) the search continues first action following the last action in $S(B,p,p_l)$. Condition 2 of the loop invariant is true because step 6 and 18 marks the last definition of y^q and y^{sbl} respectively. Condition 3 is satisfied for static program slicing because for each block X^{pbl} that is set as visited in step 29 all last definitions of all variables used in X^{pbl} are set as marked in step 29 and 30 (even if they were marked before). All these last definitions are visited and inserted in I_{bl} on the next iterations of the while-loop 25-35. Similarly for dynamic program slicing condition 3 is satisfied because each action X^k that is set as visited in step 40. In addition, last definitions of all variables used in X^k are set as marked in step 45and 46 (if they were not marked before). All these last definitions are visited and inserted in I_C on the next iterations of the while-loop 38-47. Condition 4 is satisfied because for each action X^k that is set as visited in step 40, all blocks to which X belongs is removed from R_c in step 46. Also, condition 4 is satisfied for static program slicing because for each block X^{pbl} that is set as visited in step 27, all blocks to which X belongs is removed from R_{bl} in step 33.

Proof of the theorem

A proof of theorem on the criterion based hybrid program slicing algorithm is presented Figure 23 computes correct hybrid program slices for the slicing criterion $C=(x, y^q)$.

On the termination of the repeat loop 7-15 of the algorithm, the loop invariant is combined with the termination condition. Since on the termination of the loop there are no more marked and not-visited blocks in the block trace or *LaBLock* tree trace, D_C equal to I_c and D_{bl} equals to I_{bl} . Combining this condition with the loop invariant, the following conditions are true on the exit from the repeat loop:

1. $M_C(T_x) - \bigcup_{S(B,k_1,k_2) \in \Phi_C} M_C(S(B,k_1,k_2)) = D_C$, $M_{laBlock}(Tr_{bl}) - \bigcup_{T_{Trace}(B,pbl_1,pbl_2) \in \Phi_{bl}} M_C(T_{Trace}(B_{bl},pbl_1,pbl_2)) = D_{bl}$ and $\Phi_{bl} \cap \Phi_c = \varnothing$
2. $LD(y^q) \in I_C$ and $LD(y^{sbl}) \in I_{bl}$ where $I_{bl} \cap I_c = \varnothing$
3. for all actions $X^k \in I_C$ variables $v \in U(X^k)$, $LD(v^k) \in I_C$ and for all blocks $X^{pbl} \in I_{bl}$ variables $v \in U(X^{pbl})$, $LD(v^{pbl}) \in I_{bl}$ and $I_{bl} \cap I_c = \varnothing$
4. for all actions $X^k \in I_C$, for all blocks $B \in R_c$, $X \notin N(B)$ and, for all blocks $X^{pbl} \in I_{bl}$, for all blocks $B_{bl} \in R_{bl}$, $X \notin N(B)$ and $R_{bl} \cap R_c = \varnothing$

As a result, a criterion based program slice constructed from program P by removing all the blocks that belong to $R_c \cup R_{bl}$ from it, is a criterion based hybrid program slice for the slicing criterion $C=(x, y^q)$.

4.2.5 Application of criterion based hybrid program slicing algorithm

The criterion based hybrid program slicing algorithm shown in Figure 23 is applied with two criterions on the sample program shown in Figure 13. The execution trace summary for the same is shown in Figure 24. In Figure 24, left column shows the execution trace with no static slicing criterion and the right column shows with the static criterion of *function calls*. One can easily visualize that, using static slicing for specific language constructs, the space and time can be saved. The resulting slices from the two criterions are as follows (legend: "⇒" leads to, "⇔" in between the two):

- Hybrid Criterion Static = None; ⇒ Dynamic program slicing
    - ➢ Execution Trace size : 16,577
    - ➢ Executable Slice (16 out of 32 statements):

      {1,2,3,5,6,12,17,20,21,22,24,26,27,29,30,32}
- Hybrid Criterion Static = Function calls; ⇒ Static ⇔Dynamic program slicing
    - ➢ Execution Trace size : 855
    - ➢ Executable Slice (21 out of 32 statements):

      {1,2,3,5,6,**7,8,9**,12,**14,15**,17,20,21,22,24,26,27,29,30,32}

Hybrid Criterion Static = All; ⇒ Static program slicing

- Execution Trace size : 32
- Executable Slice (23 out of 32 statements):

{1,2,3,5,6,**7,8,9**,12,**14,15**,17,20,21,22,24,**25**,26,27,**28**,29,30,32}

As stated earlier, the criterion based hybrid program slicing improves the static program slicing and minimizes the space requirements for recording the execution trace. The two extreme cases of hybrid program slicing are dynamic and static program slicing where one has no criterion another will use the entire criterion. Advantage of this type of program slicing is that users can select the slicing accuracy.

This section has introduced two hybrid program slicing techniques that are derived from the static and dynamic program slicing techniques based on the notion of removable blocks. The benefits of these algorithms are that they allow the user to have some flexibility at the same time improving the accuracy of the static program slicing.

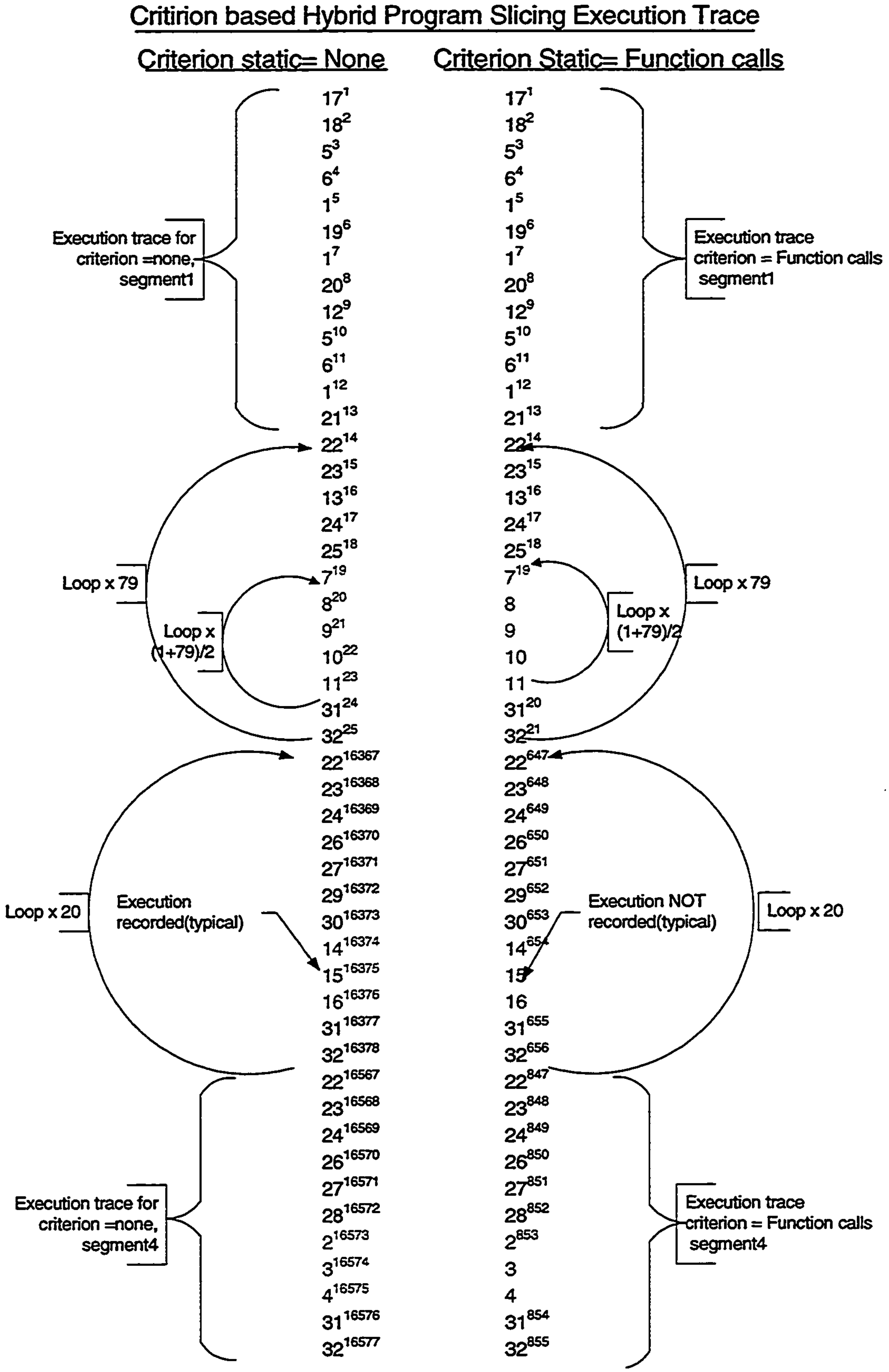


Figure 24: Condensed execution trace for sample program in Figure 13

5. Integration with MOOSE

5.1 MOOSE a comprehension framework

The MOOSE (Montreal Object-Oriented Slicing Environment) project was developed to provide an open software comprehension and maintenance framework [45]. The MOOSE framework is developed using a program slicing tool presented in [31,32]. It provides a platform for the development of advanced program slicing algorithms, slicing related features, applications and visualization techniques for both functional and object-oriented programs. The motivation for the MOOSE project is to provide an open environment that supports a variety of cognitive, visualization and algorithmic comprehension models to guide users during various program comprehension tasks. For example, understanding and analysis of existing source code the comprehension of program executions, etc. Providing higher levels of visual abstraction might not be enough to guide programmers during the complex tasks such as software comprehension of large software systems [5,21]. Most program comprehension tools represent more a collection of somewhat independent tools that provide certain analysis or visual abstraction approaches. Users (in particular novice users) are frequently confronted with these tools in significant initial learning curve caused by the large set of less intuitive functions and their associated information. Within the software engineering community, well-known concepts of good software design are module cohesion and module coupling. In the MOOSE environment we try to overcome limitations of current comprehension tools with respect to their functionality and learnability, by applying the concepts of coupling and cohesion on the functional level of program comprehension tools. One of the MOOSE design goal is to maximize the cohesion and minimize the coupling of the available functionality within the tool. Functional cohesion means a collection of tools that form from a user and task perspective a set of coherent functions that provide users with the functionality required to master a particular task and its associated information (Figure 25).

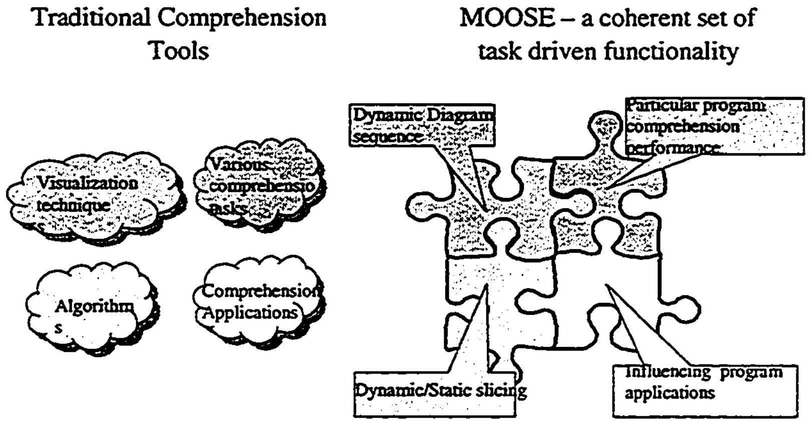


Figure 25: Task-oriented functional cohesion from a user perspective

5.1.1 MOOSE - Architecture

The MOOSE architecture (Figure 26) is based on five major components: (1) task and user centered approach that will guide users during comprehension of specific tasks, (2) an algorithmic framework, providing analysis and metric functionality, (3) an application framework that provides a set of applications supporting various comprehension tasks, (4) the visualization support, and (5) a underlying repository that provides a communication and interaction interface among all the parts of the environment.

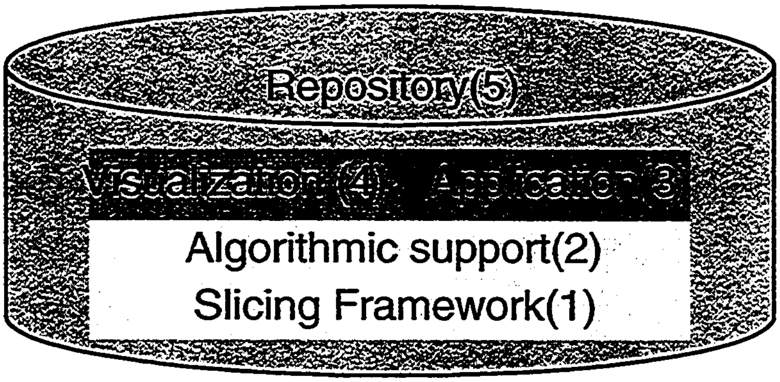


Figure 26: MOOSE program comprehension framework

The MOOSE environment was designed with two major goals in mind. The first goal was to provide a suite of tightly integrated tools with a set of coherent functionality. The second goal was to create an open environment that can easily be extended with new tools, algorithms, and applications to meet future demands. These goals are achieved by creating a general framework that consists of several sub-frameworks as illustrated in Figure 27.

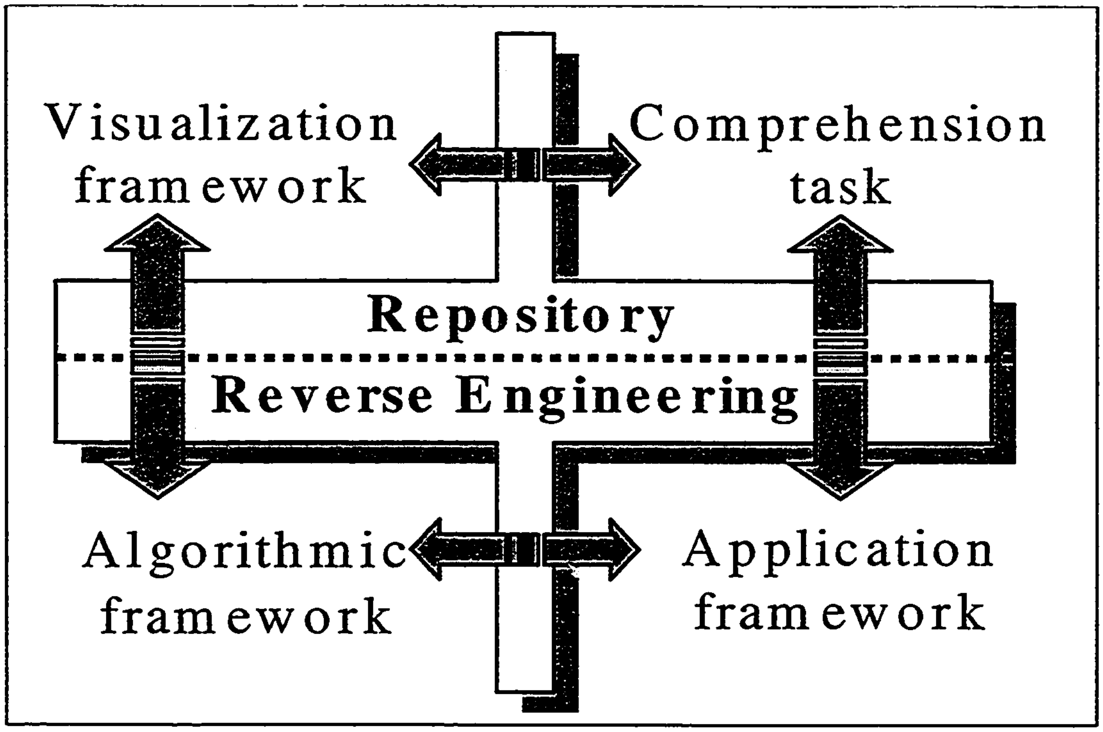


Figure 27: The open MOOSE architecture with sub frameworks

The Figure 28 shows the abstract system design of the MOOSE framework. The system design allows addition of modules without having to modify the total system as they exchange the data using an *adapter*. The new modules such as static and hybrid program slicing algorithms need to use *adapter* for external data storage and retrieval.

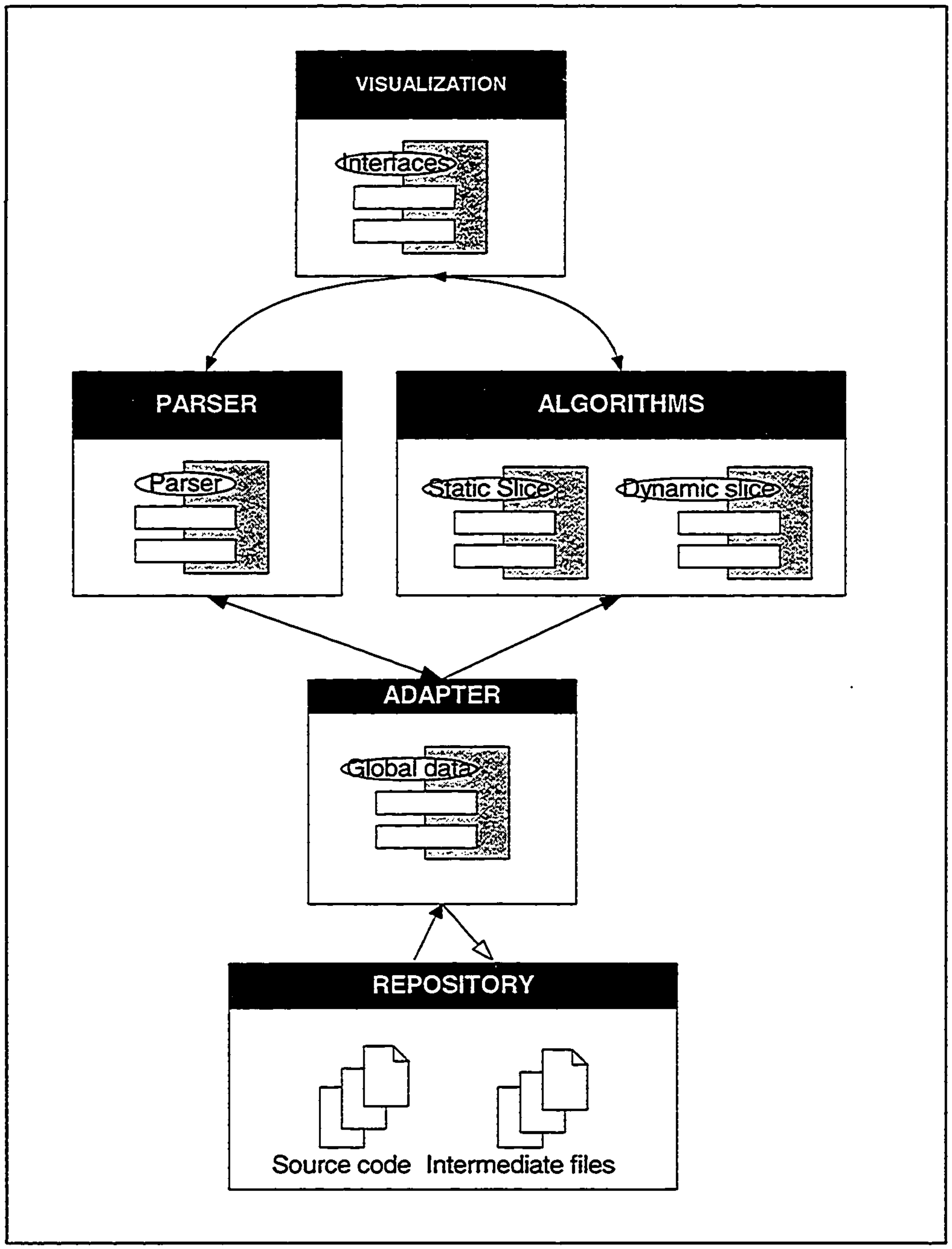


Figure 28: MOOSE System design

5.2 Analytical analysis

A few preliminary tests were carried out with basic hybrid program slicing algorithms using various sample programs to illustrate the benefits of the approach. The following table shows a sample of the data collected.

Program	Number of executable statements	Percentage of reduction after static slicing	Percentage of reduction after dynamic slicing
Program 1	32	19	50
Program 2	90	40	70

Figure 29: Comparative data on sample programs

As anticipated, the preliminary tests showed hybrid program slicing run time is more than the static program slicing, which is a small price to pay for better accuracy. The hybrid program slice run time is much lower than dynamic program slicing because many non-contributing statements have been removed using static program slicing.

However, further analytical experiments have to be carried out to compare the hybrid program slicing algorithm with static and dynamic program slicing. Through this analytical analysis, properties are identified which will allow further study the slicing algorithms in the context of their accuracy, limitations, time and space complexity and behavior for different types of programs and program executions. The properties identified for the analytical analysis is following:

Correctness

Correctness of the slice is defined for each of the language constructs that are handled properly by the algorithm. This property has to be tested with sample programs with different language constructs, for example conditional statements, loops, class constructs etc. for the proposed algorithms with the existing algorithms.

Accuracy

The goal of slicing is to compute a smallest executable subprogram from the original program. This property is referred to as *accuracy* of the program slicing algorithm. Again, this property has to be tested with sample programs with algorithm as well as other existing algorithms.

Time complexity

Time complexity is dependent on the execution length and size of the program to be sliced. This property can be analytically verified using the computation time for different algorithms with same slicing criterion.

Space complexity

Space complexity is dependent on the amount data used for the analysis in any algorithm at any one time. This property can be verified by memory requirements during the computation of slice using different algorithm with the same slicing criterion.

Further experiments needs to be carried out in the above category with the same conditions across the experiments. In other words, the comparison with different algorithms shall use the same sample program, slicing criterion and where applicable same execution length. This data could be useful in optimizing the slicing algorithms further to use with the MOOSE framework.

Program	Computation Time for Hybrid slicing	Computation Time for static slicing	Computation Time for dynamic slicing
Program1	*X1* seconds	*Y1* seconds	*Z1* seconds
...	...	...	....
Programx	*Xx* seconds	*Yx* seconds	*Zx* seconds

Figure 30: Computation time for slicing the sample programs

Program	Memory resources for Hybrid slicing	Memory resources for static slicing	Memory resources for dynamic slicing
Program1	*X1* kB	*Y1* kB	*Z1* kB
...	...	...	....
Programx	*Xx* kB	*Yx* kB	*Zx* kB

Figure 31: Memory resources for slicing the sample programs

Using the above information, MOOSE user can determine the particular type of slicing for the particular use case based on the time and space availability.

The above are only few tests which are required for basic understanding of pros and cons of the specific algorithmic support. However, different types of tests are to be carried out in addition to the above depending on the usability of MOOSE.

6. Conclusions and future work

In this thesis, two new general program slicing algorithms based on the notion of removable blocks are presented. The representation of removable blocks in the form of a syntax tree simplifies the visual information as it shows the logical tree structure with clear scope of each block. The representation provides simplified automated inclusion and exclusion of blocks based on their position relative to each other.

The thesis also introduces two hybrid program slicing approaches that improve the performance and usability of MOOSE framework. As part of the current implementation of the hybrid program slicing framework, the hybrid program slicing algorithms combines and utilizes commonalties among general dynamic and static program slicing algorithm.

Both algorithms compute correct program slices for all language constructs found in major object-oriented programming languages, e.g., polymorphism, inheritance, late binding, exception handling, local and global variables. A proof, based on the theorems and lemmas presented in [30], is presented showing that both algorithms compute correct and executable program slices.

The results from our preliminary experimental analysis show encouraging prospects for hybrid program slicing and the MOOSE framework in general.

Future work

As part of the future work, it is proposed to include *criterion-based hybrid slicing algorithm* in MOOSE framework to extend its algorithmic support. New slicing related concepts, as well as new visualization techniques should be derived to take advantage of the algorithms. The present results are still in preliminary experimentation phase and a more detailed usability and experimental analysis shall be conducted. In addition, integration of *forward* program slicing algorithms within hybrid slicing framework has to be developed to investigate additional usability aspects of MOOSE environment.

Current trends in the software engineering community are towards improving the usability of software systems. This is mainly due to the inability shown by current software developers to produce software that is easy to use and provides the required functionality. Desmarais [9] shows that only 50% (±20%) of the services of most software applications is ever mastered. The other half of the services are either not useful to specific tasks, or users have never had the time to master it, or users ignore their existence. Many CASE tools have not performed as expected due to the inability of the products to be consistent with the needs of CASE users and system developers. Object-oriented software development utilizes new design methodologies. The use of the term "quality in use" implies that it is necessary to take into account human-centered issues in evaluating software products. "Quality in use" is the extent to which an entity satisfies stated and implied needs when used under specified conditions [46,48,49]. Therefore, it is proposed that not only planning to investigate the usability of the presented concepts but also to derive new slicing related features based on the user feedback with the tool.

References

1. Agrawal, H., "Towards automatic debugging of computer programs", *Technical Report SERTC-TR-40-P,* Purdue University, 1989

2. Agrawal, H. and Horgan, J, "Dynamic program slicing", In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 25(6), pp. 246-256, 1990.

3. Agrawal, H., DeMillo, R., and Spafford, E., "Debugging with dynamic slicing and backtracking", Software – Practice and Experience, 23(6), pp. 589-616, 1993.

4. Agrawal, H., "On Slicing programs with jump statements", In proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, pp. 112-135, 1994.

5. Binkley D. and Gallagher K., "Program Slicing", Adv. in Computers, 43, Academic Press, pp. 1-52, 1996.

6. Chen J.L., Wang F.J., and Chen Y.L., "Slicing object oriented Programs", Proceedings of the APSEC'97, pp. 395-404, Hongkong, China, December 1997.

7. Cheng, J., "Slicing concurrent programs a graph-approach", In Proceedings of the First International Workshop on Automated and Algorithmic Debugging (1993), P. Fritzson, Ed., Vol. 749 of Lecture Notes in Computer Science, Springer-Verlag, pp. 232-245, 1993

8. Choi, J.-D., Miller, B., and Netzer, R., "Techniques for debugging parallel programs with flowback analysis", ACM Transactions on Programming Languages and Systems, 13(4), pp. 491-530, 1991.

9. Desmarais, M.C., Liu, J. "Exploring the applications of user-expertise assessment for intelligent interfaces". In Proceedings of InterCHI'93, Bridges between worlds (Amsterdam, 24-29 April), pp. 308-313, 1993.

10. Duesterwald, E., Gupta, R., and Soffa, M., "Distributed slicing and partial re-execution for distributed programs", In Proceedings of the fifth workshop on Languages and Compilers for Parallel Computing, New Haven, Connecticut, pp. 329-337, 1992.

11. Ferrante, K., Ottenstein, K, and Warren J. "The Program Dependence Graph and its use in Optimization", In ACM Transactions on Programming Languages and Systems, 9(5), pp.319-349, 1987.

12. Gallagher, K. and Lyle, J., "Using program slicing in software maintenance", IEEE Transactions on Software Engineerig, 17(8), pp. 751-761, August 1991

13. Gopal, R., "Dynamic program slicing based on dependence relations", In Proceedings of the Conference on Software Maintenance, pp. 191-200, 1991.

14. Gupta, R., Harrold, M., and Soffa, M., "An approach to regression testing using slicing", In Proceedings of the Conf. on Software Maintenance, , pp. 299-306, 1992.

15. Gupta, R., Soffa, M. and Howard J., "Hybrid Slicing: Integrating Dynamic Information with Static Analysis", ACM Transactions on Software Engineering and Methodology, 6(4), pp. 370-397, October 1997.

16. Harman M. and Gallagher K., editors, Journal of Information and Software Technology Special Issue on Program Slicing, volume 40. Elsevier, 1998.

17. Hart, J.M, "Experience with Logical code analysis in software reuse and reengineering", In AIAA computing in Aerospace, 10, , pp. 1243-1262, San Antonio, Texas, March 28-30, 1995.

18. Hendley R., et al.: "Case Study - Narcissus: Visualizing Information", Proceedings of the IEEE Information Visualization 95, pp. 90-96, 1995.

19. Horwitz S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs", ACM Transactions on Progr. Languages and Systems, 12(1), pp. 26-61, 1990.

20. Horwitz, S. and Reps, T., "The use of program dependence graphs in software engineering", In Proceedings of the 14th Int. Conference on Software Engineering, pp. 392-411, Melbourne, Australia, 1992.

21. Kamkar M., "Interprocedural Dynamic Slicing with Applications to Debugging and Testing", Ph.D. Thesis, Linköping University, 1993.

22. Kamkar M., Fritzson, P., and Shahmehri, N., "Three approaches to interprocedural dynamic slicing", Microporcessing and Microprogramming, (38), pp. 625-636, 1993.

23. Korel, B. and Laski, J., "Dynamic program slicing", In. Proc. Letters, 29(3), pp. 155-163, Oct. 1988.

24. Korel, B., "PELAS – Program Error Locating Assistant System", IEEE Trans. on Software Engineering, 14(9), pp. 1253-1260, Sept. 1988.

25. Korel, B. and Laski, J., "Dynamic program slicing", Information Proc. Letters, 29(3), pp. 187-195, 1990.

26. Korel, B. and Ferguson, R., "Dynamic slicing of distributed programs", Applied Mathematics & Computer Science Journal, 2(2), pp. 199-215, 1992.

27. Korel, B., “Identifying faulty modifications in software maintenance”, Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, pp. 341-356, Linköping, Sweden, 1993.

28. Korel, B. and Yalamanchili, S., “Forward Derivation of Dynamic Slices”, Proc. of the Intern. Symposium on Software Testing and Analysis, pp. 66-79, Seattle, 1994.

29. Korel,B., “Computation of Dynamic Slices for Programs with Arbitrary Control-flow”, The 2nd International Workshop on Automated and Algorithmic Debugging, pp. 1-41, St. Malo, France, 1995

30. Korel, B., “Computation of dynamic slices for unstructured programs”, IEEE Transactions on Software Engineering, 23(1), pp. 17-34, 1997.

31. Korel B. and Rilling, J., “Application of Dynamic Slicing in Program Debugging”, Third International Workshop on Automated Debugging (AADEBUG'97), pp. 59-74, Linköping, Sweden, May 1997.

32. Korel, B. and Rilling, J., “Program Slicing in Understanding of Large Programs”, Proceedings of the 6th IWPC ‘98, pp. 145-152, Ischia, Italy, June 1998.

33. Korel, B. and Rilling, J., “CASE and Dynamic Program Slicing in Software Maintenance”, Special issue of the International Journal of Computer Science and Information Management, (June 1998)

34. Krishnaswamy A., "Program slicing: An application of object-oriented Program Dependency Graphs", Technical report TR94-108, Cs Dep., Clemson University, 1994.

35. Kung D. Gao J. et al., “Developing an object-oriented software testing and maintenance environment”; In Com. of the ACM, Vol. 38, Issue 10 (1995), pp. 75-87.

36. Larsen L.D. and Harrold M.J., "Slicing Object oriented software", Proceeding of the 18th International conference on Software engineering, March,1996.

37. Law R.C.H., "Object-Oriented Program Slicing" Ph.D. thesis, University of Regina, Regina, Canada, 1994.

38. Lyle, J. and Weiser, M., “Experiments on slicing-based debugging tools”, Proc, of the 1st Conference on Empirical Studies of Progr.”, pp. 187-197, 6/ 1986.

39. Mayerhauser A, Vans A. M., “Program Understanding Behavior During Adaptation of Large Scale Software”, Proc. of the 6th Intl. Workshop on Program Comp. IWPC ‘98, pp. 164-172, Ischia, Italy, June 1998.

40. Ottenstein, K., and Ottenstein, L., "The program dependence graph in a software development environment", In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN 19(5),pp.177-184, 1984

41. Reps, T., and Horwitz S., "Semantics-based program integration", In Proceedings of the Second ACM European Symposium on Programming (ESOP'88), pp. 133-145, Nancy, France, March 1988.

42. Reps, T. and Bricker, T. "Semantics-based program integration illustrating interference in interfering versions of programs", In Proceedings of the Second International Workshop on Software Configuration Management, pp. 46-55, Princeton, New Jersey, Oct. 1989.

43. Richner T. and Stéphane Ducasse, "Recovering High - Level Views of Object - Oriented Applications from Static and Dynamic Information", In Proc. of ICSM'99, September, IEEE Computer Soc. Press, pp. 13-22, 1999.

44. Rothermel G. and Harrold, M.J., "Selecting tests and identifying test coverage requirements for modified software", In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 169-184, August 1994.

45. Rilling J, "Investigation Of Dynamic Slicing And Its Application In Program Comprehension", Ph.D. Thesis; Illinois Institute of Technology, July 1998.

46. Rilling J and Seffah A, "Enhancing the Usability and Learnability of Software Visualization Techniques through Task Wizards and Software Agents", 2001 International Conference on Imaging Science, Systems, and Technology (CISST'200), June 25-28, 2001, Las Vegas, Nevada, USA.

47. Schoenig S. and Ducass'e. M. "A hybrid backward slicing algorithm producing executable slices for Prolog". In Proceedings of the 7th Workshop on Logic Programming Env. Pages 41–48, Portland, USA, December 1995.

48. Seffah Ahmed. "Training Software Developers in Critical Skills" IEEE Software Magazine, 6/1999.

49. Seffah A and Rilling, J "Investigating the Relationship between Usability and Conceptual Gaps for Human-Centric CASE Tools", IEEE Symposium on Human-Centric Computing Languages and Environments, Stresa, Italy, September 5-7, 2001

50. Shimomura T., "The program slicing technique and its application to testing, debugging and maintenance", Jour. of IPS of Japan, 9(9), pp. 1078-1086, Sept. 1992.

51. Steindl,C. *Intermodular slicing of object-oriented programs.* In International Conference on Compiler Construction (CC'98), 1998

52. Steindl, C., “Static Analysis of Object-Oriented Programs”, 9th ECOOP Workshop for Ph.D. Students in OO-Programming, Lisbon, Portugal, June 14, 1999.

53. Storey, M. -A.D.; Wong, K.; Muller, H.A., "How do program understanding tools affect how programmers understand programs?" Proceedings of the Fourth Working Conference on Reverse Engineering, p. 12-21, Netherlands, 6-8/10/97.

54. Tip F., “A survey of program slicing techniques”, Journal of Progr. Languages, 3(3), pp. 121-189, 9/1995.

55. Tip.F, Choi J.D., Field J. and Ramalingam G. "Slicing Class Hierarchies in C++," Proc. of the 11th Annual conference on Object-Oriented Programming, systems, Languages, and Applications, pp.179-197, October,1996.

56. Weiser, M., “Programmers use slices when debugging”, Communications of ACM, 25, pp. 446-452, 1982.

57. Weiser, M., “Program slicing”, IEEE Transactions on Software Engineering 10(4), pp. 352-357, 1984

58. White, L. and Leung, H., “Regression testability”, IEEE Micro, pp. 81-85, April 1992.

59.Zhao.J, Cheng J, and Ushijima K, "Static Slicing of Concurrent Object-Oriented Programs", Proceedings of the 20th IEEE Annual International Computer Software and Applications conference , pp.312-320, August, 1996, IEEE Computer Society Press.

60. Zhao.J, " Dynamic Slicing of Object-Oriented Programs," Technical-Report SE-98-119, pp.17-23, Information Processing Society of Japan, May 1998.

Appendix A

General dynamic program slicing algorithm

The algorithm presented in [30,45] identifies a set of removable blocks R_C. In the first step of the algorithm, the program is executed and its execution trace is recorded up to execution position q. All actions in the execution trace are set as unmarked and not visited. Set R_C initially contains a set of all blocks in the program. In step 4, a set of contributing actions I_C is initialized as an empty set. In step 5 the algorithm identifies and marks the last definition Y^p of variable y^q, this step (procedure) is presented in more detail in lines 53-57. The algorithm iterates in the repeat loop 6-10 until all marked actions are visited. There are three major steps inside of the repeat loop.

In step 7, the algorithm identifies actions that contribute to the computation of y^q. In step 8, for the given set of contributing actions, the algorithm identifies non-contributing actions by finding a set of block traces. The more non-contributing actions can be identified, the smaller dynamic slices may be computed. In step 9, all the remaining actions, i.e., actions that are not identified as contributing or as non-contributing actions, are marked. These actions are visited on the next iteration of the repeat loop. The process of identifying contributing and non-contributing actions continues in the repeat loop until all actions are classified as contributing or non-contributing actions.

In what follows, a more detailed description of the major steps of the dynamic slicing algorithm of is presented. In step 7, the algorithm identifies contributing actions. This step (procedure) is presented in more detail in lines 12-23. The major component of this while loop are that during each iteration a marked and not visited action X^k is selected and set as visited in lines 15 and 16. In addition, X^k is inserted into I_C. All last definitions of all variables used in X^k are identified and marked in line 17-19. In line 20-21, all blocks that contain node X are removed from Rc. The while loop iterates until all marked actions are visited. In step 8 the algorithm identifies non-contributing actions for the set I_c of contributing actions (the actions are set as visited) and for the set of blocks R_C. This step (procedure) is presented in more detail in lines 24-42. The goal of this step is to find as many non-contributing actions as possible. The more non-contributing actions can be identified, the smaller dynamic slices may be computed. The procedure identifies non-contributing actions by finding a set Φ_C of block traces for the set of blocks R_C. All block traces of Φ_C contain only unmarked actions. The procedure explores the execution trace from the beginning looking for actions that are not set as visited. If such an action is found then for this action the procedure tries to identify block trace S $(B, p, p1)$ of block B that belongs to R_C by finding r-entry and r-exit of B at position p and p_1, respectively. If such a block trace is found then it is inserted into Φ_C. Since all actions in block trace S $(B, p, p1)$ are non-contributing actions, the algorithm continues the search for non-contributing actions (block traces) starting from position p_1. This process of identifying block traces of blocks that belong to R_C continues until the end of execution trace is reached at position q. In step 9, all actions that have not been identified as contributing or as non-contributing actions are marked in step 9 because they are considered as contributing actions. This step (procedure) is presented in more detail in lines 43-47

Algorithm

Input: a slicing criterion $C=(x, y^q)$
Output: a dynamic program slice for C
Legend:

T_x	execution trace up to position q
Φ_C:	Set of block traces
Rc:	Set of blocks
I_C:	Set of contributing actions (default is visited)
N_C	non-contributing action
C_C	contributing action

```
Execute program $P$ on input $x$ and record execution trace $T_x$ up to position $q$
Initialize $Rc$ to a set of all blocks in program $P$
Set all actions in $T_x$ as not marked and not visited.
$I_C = \emptyset$
Find last definition $y^p$ of $y^q$ and set $y^p$ as marked
repeat
    Identify $C_C$
    Identify $N_C$
    Mark remaining actions
until every marked action in $T_x$ was visited
Create dynamic program slice by removing all blocks $\in Rc$
function Identify $C_C$
while (contributing and not visited action in $T_x$)
    Select a contributing and not visited action $X^k$ in $T_x$
    Set $X^k$ as a visited action
    $I_C = I_C \cup \{X^k\}$
    for all variables $v \in U(X^k)$ do
        Identify + mark last def of $v$ as contributing action
    end-for
    for all blocks $B \in Rc$ do
        if $X \in N(B)$ then $Rc = Rc - \{B\}$
end-while
end Identify $C_C$
function Identify $N_C$
```

Figure 32: General dynamic program slicing algorithm [30](continued)

```
$\Phi_C = \varnothing$
$p = 1$
do
  Let $X^p$ be an action at position $p$ in $T_x$
  if $X^p \notin I_c$ then
    Let $Y^t \in I_c$ be the closest $C_c$ from position $p$ such that $t > p$
    if there exists block $B \in Rc$ such that
      (1) $B$ has an r-entry at position p, and
      (2) there exists no inheritance
      (3) B has an r-exit between $p$ and $t$ then
        Let $p_1$ be the closest position of an r-exit of B from position $p$, where $p < p_1 < t$
        $\Phi_C = \Phi_C \cup \{S(B,p,p_1)\})$
        $p = p_1 - 1$
    end-if
  end-if
  $p := p + 1$
while $p \geq q$
end Identify $N_c$
function Mark remaining actions
for all actions $X^k$ that are not identified $C_c$ nor as $N_c$
    Set $X^k$ as marked
end-for
end Mark remaining actions
function Identify inheritance
if current class is derived then
    set all $X^{base}$ classes as contributing
end-if
end Identify-inheritance
function Find last definition
  Find last definition of variable $y^p$ in global scope
  If not found last definition then
    search last definition of $y^p$ within local scope
end Identify-last-definition
```

Figure 32: General dynamic program slicing algorithm [30]

One of the extensions presented in [45] for generic dynamic slicing algorithm is the identification of the last definition of variables and their scopes and the identification of inherited classes. As part of the object-oriented languages constructs, additional variable types and their variable scopes have to be considered. The function in line 53-57 not only identifies the scope of variables, it also has to take into consideration the variable types, e.g.; user defined, passed by value, passed by reference, etc. The function *identify inheritance* in line 48-52 handles the identification of inheritance and multiple inheritance cases as described earlier, by marking all classes from which the current class inherits as non removable.

It should be mentioned that the presented generic dynamic slicing algorithm also computes correct dynamic slices for the programming construct like nested function calls and recursion.

Proof of the correctness of generic dynamic slicing algorithm

The same theorems and lemmas presented in [30] to prove the correctness of the algorithm also applies to the following generic dynamic slicing algorithm. The following section provides a summary of the theorems and lemmas.

Theorem:

The generic dynamic slicing algorithm presented in Figure 32 correctly computes a dynamic program slice for the slicing criterion $C=(x,y^q)$.

In order for the generic dynamic slicing algorithm to compute correct dynamic slices the algorithm has to satisfy Lemma 1 and 2 presented in [30] . The same formal proof might be applied for the general algorithm.

We create the following loop invariant, presented at line 10 of the algorithm in Figure 32.

1. D_C is a set of all actions set as marked or visited in T_X.

$M(T_X) - \bigcup M(S(B,k_1,k_2)) = D_C$, where $S(B,k_1,k_2) \in \Phi_C$

2. $LD(y^q) \in I_C$

3. for all actions $X^k \in I_C$ variables $v \in U(X^k)$, $LD(v^k) \in I_C$,

4. for all actions $X^k \in I_C$, for all blocks $B \in R_C$, $X \notin N(B)$.

Proof of the loop invariant

The loop invariant at line 10 is true for each of the loop iterations. Notice that I_C in the algorithm is a set of all actions that are set as visited, whereas D_C is a set of all actions that are set as marked or visited. Therefore, I_C is a subset of D_C. Steps 8 and 9 satisfy condition 1 of the loop invariant. Step 8 ensures that all identified block traces that are included in Φ_C, exclude actions set as visited (lines 28, 29 and 35). Step 9 sets all actions that are identified as either contributing or as non-contributing actions as marked. Notice that all block traces in Φ_C are disjoint because when a block trace $S(B,p,p_1)$ is found (in lines 31-36), the search continues from the first action following the last action in $S(B,p,p_1)$. In addition, only traces of blocks that belong to R_C are considered (line 31). Condition 2 of the loop invariant is true because step 5 marks the last definition of y^q. Condition 3 is satisfied because for each action X^k that is set as visited in step 15 all last definitions of all variables used in X^k are set as marked in step 17&18 (if they were not marked before). All these last definitions are visited and inserted in I_C on the next iterations of the while-loop 13-22. Condition 4 is satisfied because for each action X^k that is set as visited in step 15, all blocks to which X belongs is removed from R_C in step 21.

Proof of the theorem

A proof of theorem on the generic dynamic slicing algorithm is presented in Figure 32 computes correct dynamic slices for the slicing criterion $C=(x,y^q)$.

On the termination of the repeat loop 6-10 of the algorithm, the loop invariant is combined with the termination condition. Since on the termination of the loop there are no more marked and not-visited actions in the execution trace, D_C equals to I_C ($D_C = I_C$). Combining this condition with the loop invariant, the following conditions are true on the exit from the repeat loop:

1. $M(T_x) - \bigcup M(S(B,k_1,k_2)) = I_C$ where $S(B,k_1,k_2) \in \Phi_C$
2. $LD(y^q) \in I_C$,
3. for all actions $X^k \in I_C$ and variables $v \in U(X^k)$, $LD(v^k) \in I_C$
4. for all actions $X^k \in I_C$, for all blocks $B \in R_C$, $X \notin N(B)$.

As a result, a dynamic program slice constructed from program P by removing from it all blocks that belong to R_C is a dynamic program slice for the slicing criterion $C=(x,y^q)$.