# A Criterion Based Hybrid Slicing Algorithm for Object-Oriented Programs

Bin Wu

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
Montreal, Quebec, Canada

August, 2003

*Your file Votre référence*
*ISBN: 0-612-83922-2*
*Our file Notre référence*
*ISBN: 0-612-83922-2*

# Abstract

## A Criterion Based Hybrid Slicing Algorithm for Object-Oriented Programs

**Bin Wu**

Program slicing, a program reduction technique that is used to simplify programs by removing non-relevant parts with respect to a slicing criterion, is gaining more and more attention in the software comprehension and maintenance community. In this research, we provide an overview of program slicing approaches, which are categorized into static, dynamic and hybrid slicing. Moreover, we present our own hybrid slicing algorithm that is based on the notion of removable blocks and computes executable slices for Java programs. The algorithm is implemented as part of the CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) project. One advantage of our hybrid slicing algorithm is that it takes user inputs into account to customize the precision of the algorithm accordingly to a user's needs. As a result, our hybrid slicing algorithm provides the user with the ability to take advantage of both static and dynamic slicing, by avoiding at the same time their disadvantages.

# Acknowledgments

I would like to express my sincere gratitude to my supervisor Dr. Juergen Rilling for his support, guidance, patience, and valuable insight, which have made the completion of my thesis possible.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Software maintenance is a major part of the software development process. It accounts for about 60% of the costs in the lifetime of the software system. In software maintenance, however, understanding existing software is very difficult, accounting for about half of a maintenance programmer's time and effort [Sch95]. Program slicing can simplify the program by removing parts of a program that are not relevant with respect to a particular slicing criterion. In recent years program slicing has become an important aide in software comprehension.

Through over two decades of research, many approaches to program slicing have been proposed. In general, they can be classified into static program slicing and dynamic program slicing. Static program slicing based only on statically available information was first introduced by Weiser [Wei82, Wei84] in 1979. The approach uses dataflow equations to find statements relevant to a set of variables at a point of interest. In 1988, Korel and Laski [Kor88] introduced dynamic program slicing, which refined static program slicing by involving dynamic information derived from program execution based on particular inputs.

There are limitations in both static program slicing and dynamic program slicing. On the one hand, static program slicing is not precise because it preserves the program behaviour for the set of all inputs and has to consider all possibilities, especially for the object-oriented program that involves polymorphism and dynamic bindings. On the other hand, dynamic program slicing incurs a high run-time overhead either to record the dynamic behaviour (execution) and/or to analyze every executed statement. The space and time complexity becomes in particular an issue for long running programs with a very large number of executed statements. Hybrid program slicing algorithms try to combine the advantages of static and dynamic program slicing, by computing a hybrid slice that is either as precise or more precise than the static slice and less costly than the dynamic slice.

In this research, a hybrid program slicing algorithm for Java program is proposed and implemented as part of the CONCEPT project. The remainder of the paper is organized as follows: Section 2 introduces in more detail some general background about program slicing, including existing static, dynamic and hybrid program slicing approaches. Section 3 presents our hybrid program slicing algorithm and describes its static and dynamic parts. In section 4, implementation issues of the algorithm in the context of the CONCEPT project are presented. Section 5 concludes and discusses directions for future work.

## 2 Background of Program Slicing

The concept of a *program slice* was first presented by Weiser in 1979. Weiser [Wei82, Wei84] defined a slice as a reduced, executable program that preserves the original behaviour of the program with respect to a subset of variables of interest at a given program point. That is, a program slice consists only of these parts of a program that can potentially affect the values computed at a point of interest. Such a point of interest, that combines a program location and a subset of program variables of interest, is called *slicing criterion*. Usually, a slice criterion is represented by a pair (V, *n*), where V is a set of variables and *n* is a point of interest within the program. *Program slicing* itself, corresponds to the task of computing a program slice.

```
0     public class Sample{
1       private int val;
2       public Sample(int x){
3           val = x;
        }
4       public int getVal(){
5           return val;
        }
6       public void setVal(int x){
7           val = x;
8           return;
        }
9       public void min(int x, int y){
10          if(x < y){
11            setVal(x);
12            y = x;
13          }else{
14            setVal(y);
15            x = y;
            }
16          System.out.println("Min is: " + x);
        }
17      public static void main(String[] args){
18          int a = new Integer(args[0]).intValue();
19          int b = new Integer(args[1]).intValue();
20          Sample s = new Sample(a);
21          int i = 0;
22          while(i < 2){
23            s.min(a, b);
24            a = s.getVal();
25            System.out.println(a);
26            a = a + 10;
27            b = b - 10;
28            i++;
         }
        }
      }
```

Figure 1. Sample program for introducing program slicing

An example of such a program slice computation is shown in Figure 1. Within this simple program, a slice with respect to a slicing criterion (a, 25) is computed. The slice is shown in a bold typeface. In general, a program slice can be executable or non-executable, with executable referring to the fact, that the program slice can be recompiled and re-executed. A non-executable slice on the other hand corresponds to a set of statements, which might not guarantee a semantically and syntactically correct program and therefore it cannot be re-compiled and re-executed. For the scope of this research we are focusing on the computation of executable program slices, which is easier to be verified, at least from a syntactical and logical viewpoint, since the output of the executable slice (with respect to the slicing criterion) has to be identical to the original program. Since Weiser's introduction of the notion of program slicing, various types of other program slices, as well as an even larger number of algorithms to compute program slicing have been proposed. Generally, there are two major kinds of program slicing algorithms: static and dynamic program slicing. A static slicing algorithm computes a static slice based only on statically available information, considering all possible value inputs; while a dynamic slice is to obtain by considering only statements that influence the value of variables of interest based on particular input(s).

## 2.1 Static Program Slicing

The first published definition of program slicing was given by Mark Weiser, who introduced the concept of static program slicing [Wei82] [Wei84]. He provided the following definition of a static program slice:

***Definition:*** Given is a slicing criterion as a pair C = (V, n), where V is a set of variables, and n is a point of interest within a program p; a slice of p is any program which has the same behavior as p upon the variables in V at n. A static slice includes therefore all statements that potentially affect variables in V for a set of all possible inputs at the point of interest n.

Program slicing uses methods and terminology from the program dependence theory. No matter what kind of approach to program slicing is used, data dependence and control dependence are the two basic program dependences to be analyzed by these algorithms. Ferrante et al. [Fer87] introduced the notion of control dependence to represent the relation between program entities due to control flow. Some relative and basic notions including control flow graphs, dominator and post-dominator were defined by Brandis [Bra95] and Aho et al [Aho96].

**Definition:** A *control flow graph* is a directed graph with a unique entry node START and a unique exit node STOP. The nodes in the graph correspond to statements of the program. There is a directed edge from node A to node B if control may flow from block A directly to block B. This is the case if the last statement in A is a branch to B, or when B is on the fall-through path from A. We assume that for any node N in the graph there exists a path from START to N and a path from N to STOP. If an edge is labelled T (or F), then the target node of the edge will be executed if the predicate at the origin of the edge evaluates to TRUE (or FALSE).

An example of a control flow graph based on the function "min(int x, int y)" of the sample program in Figure 1 is shown below in Figure 2.

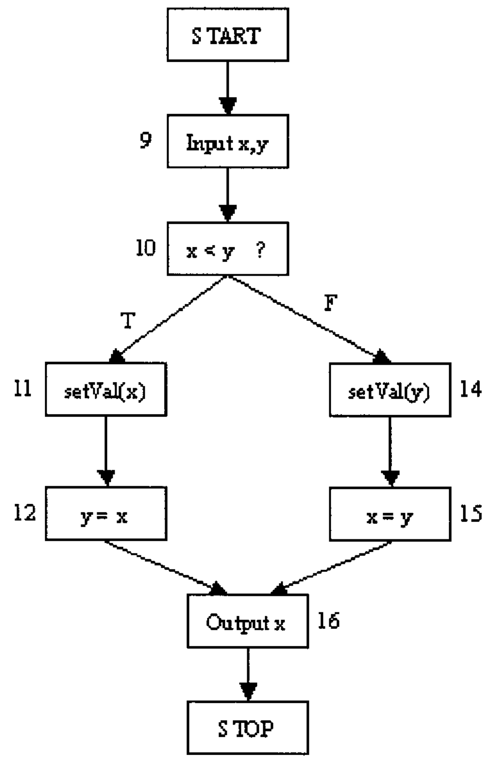


Figure 2. Control flow graph of function "min" in Figure 1

**Definition:** In a directed graph with entry node START, node A *dominates* node B, if for all paths P from START to B, A is a member of P. A is called a *dominator* of B.

In Figure 2, node 10 is a dominator of node 11.

**Definition:** In a directed graph with exit node STOP, a node A *post-dominates* node B, if for all paths P from B to STOP, A is a member of P. A is called a *post-dominator* of B.

For example, in Figure 2, node 12 is a post-dominator of node 11.

**Definition:** Let G be a control flow graph. Let A and B be nodes in G. B is *control dependent* on A if all of the following hold:

1. There exists a directed path P from A to B.
2. B post-dominates any C in P (excluding A and B).
3. B does not post-dominate A.

In Figure 2, node 12 is control dependent on node 10.

Aho et al. [Aho86] introduced the concept of data dependence based on data flow that describes the flow of the values of variables from the points of their definitions to the points where their values are used.

**Definition:** Node U is *data dependent* on node D if all of the following hold:

1. Node D defines variable x. That is, a value is assigned to variable x.
2. Node U uses x. That is, variable x is in the statement but not be assigned a value.
3. Control can reach U after D via an execution path along which there is no intervening definition of x.

In Figure 1, node 18 defines the variable *a* that is used in node 20, and there is no other definition of *a* within the path from node 18 to node 20. Therefore, node 20 is data dependent on node 18.

### 2.1.1 Dataflow Equations

Dataflow equations, based on control flow graph are used to compute consecutive sets of relevant variables for each node and were originally introduced in Weiser's algorithm [Wei84]. The slice with respect to node n and variables V can be derived by analyzing the data flow information about the sets of relevant variables. The algorithm can be described simply as follows:

```
1. Initialize the relevant sets of all nodes to be empty .
2. Insert all variables of V into relevant(n).
3. For n's immediate predecessor m, let def(m) be the set of variables that are
   defined in statement m. Compute relevant(m) as:
       relevant(m) = relevant(n) – def(m)
       if relevant(n) ∩ def(m) != {} then
           relevant(m) = relevant(m) U ref(m)
           include m into the slice
           include m's control dependence into the slice
       end
4. Work backwards in the control flow graph, repeating step 3 for m's immediate
   predecessors until the entry node is reached or the relevant set is empty.
```

In Table 1 an example for function "min()" based on the sample program in Figure 1 is shown to illustrate dataflow equations. The slicing criterions is C(x, 16) and the slice includes all nodes in bold face.

Table 1. Relevant sets for function “min” in Figure 1

| n | Statement | Ref(n) | Def(n) | Control(n) | Relevant(n) |
|---|---|---|---|---|---|
| **9** | min(int x, int y) | | x, y | | |
| **10** | if(x < y) | x, y | | | x, y |
| 11 | setVal(x) | x | | 10 | x |
| 12 | y = x | x | y | 10 | x |
| **13** | Else | | | 10 | |
| 14 | setVal(y) | y | | 13 | y |
| **15** | x = y | y | x | 13 | y |
| **16** | output x | x | | | x |

### 2.1.2 Program Dependence Graph

A Program Dependence Graph (PDG) is a program representation where nodes represent statements and edges carry information about control and data dependences. The PDG was originally defined by Ottenstein [Ott84] and later refined by Horwitz et al. and Reps[Hor88, Hor90a, Rep89, and Rep94]. The static slice of a program with respect to a variable v at node n, consisting of all nodes whose execution could possibly affect the value of variable v at node n, it can be easily constructed by traversing backwards along the edges of a program dependence graph starting at node n. The nodes, which were visited during the traversal, constitute the desired slice. According to the definition of PDG, we can get the PDG of function “min()” of the sample program in Figure 1 shown in Figure 3.

Assuming the slicing criterion is C(x, 16), we traverse backwards through all incoming edges starting at node 16. Hence we get the static slice that corresponds to a set of nodes, in this case {9,10,13,15,16}. The result is equivalent to the slice derived using Data Equations.

It should be noted that the PDG is only suited for intra-procedural analysis of a program slice because there is no representation of information between functions in the graph. For inter-procedural traversal, a System Dependence Graph (SDG) has to be used.

### 2.1.3 System Dependence Graph

A System Dependence Graph (SDG) [Hor89] is a collection of procedure dependence graphs, one for each procedure. A Procedure Dependence Graph [Fer87] represents a procedure as a graph where vertices are statements or predicate expressions, while edges represent data and control dependences between statements or expressions. Each Procedure Dependence Graph contains an entry vertex that represents an entry into the procedure. To model parameter passing, in a SDG [Lar96] associates each procedure entry vertex with a formal-in vertex for each formal parameter and a formal-out vertex for each formal parameter that could be modified by the procedure. In addition to formal-in and formal-out vertices, an SDG associates each call site in a procedure with a call vertex and a set of actual-in and actual-out vertices. An actual-in vertex corresponds

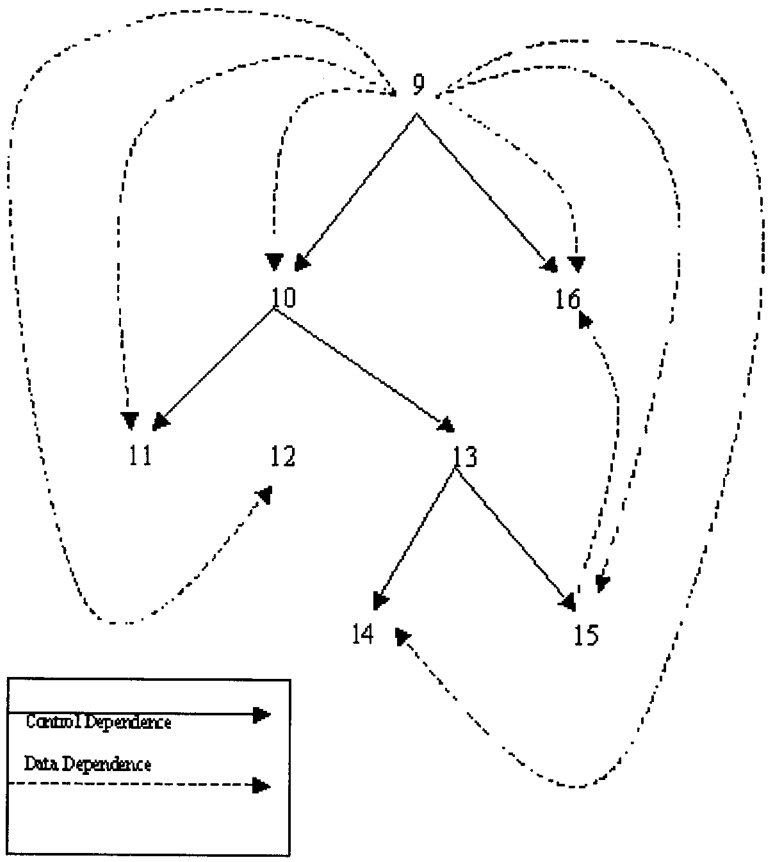


Figure 3. PDG of function "min()" of sample program in Figure 1

to an actual parameter at the call site, while an actual-out vertex is for an actual parameter that may be modified by the called procedure. Moreover, at procedure entry and call sites, global variables are treated as parameters. Thus, there are actual-in, actual-out, formal-in and formal-out vertices for these global variables.

The SDG uses a call edge to connect the call site vertex and the entry vertex of the called procedure's PDG. Parameter-in and parameter-out edges, representing parameter passing, respectively connect actual-in and formal-in vertices, and formal-out and actual-out vertices. Besides edges mentioned above, summary edges are most important for the SDG to solve the "calling context" [Tip95] problem that is hard to solve by using dataflow equations. There exist several different algorithms to compute these summary edges [Hor90b, Rep94]. A SDG that represents the sample program of Figure 1 is shown below in Figure 4.

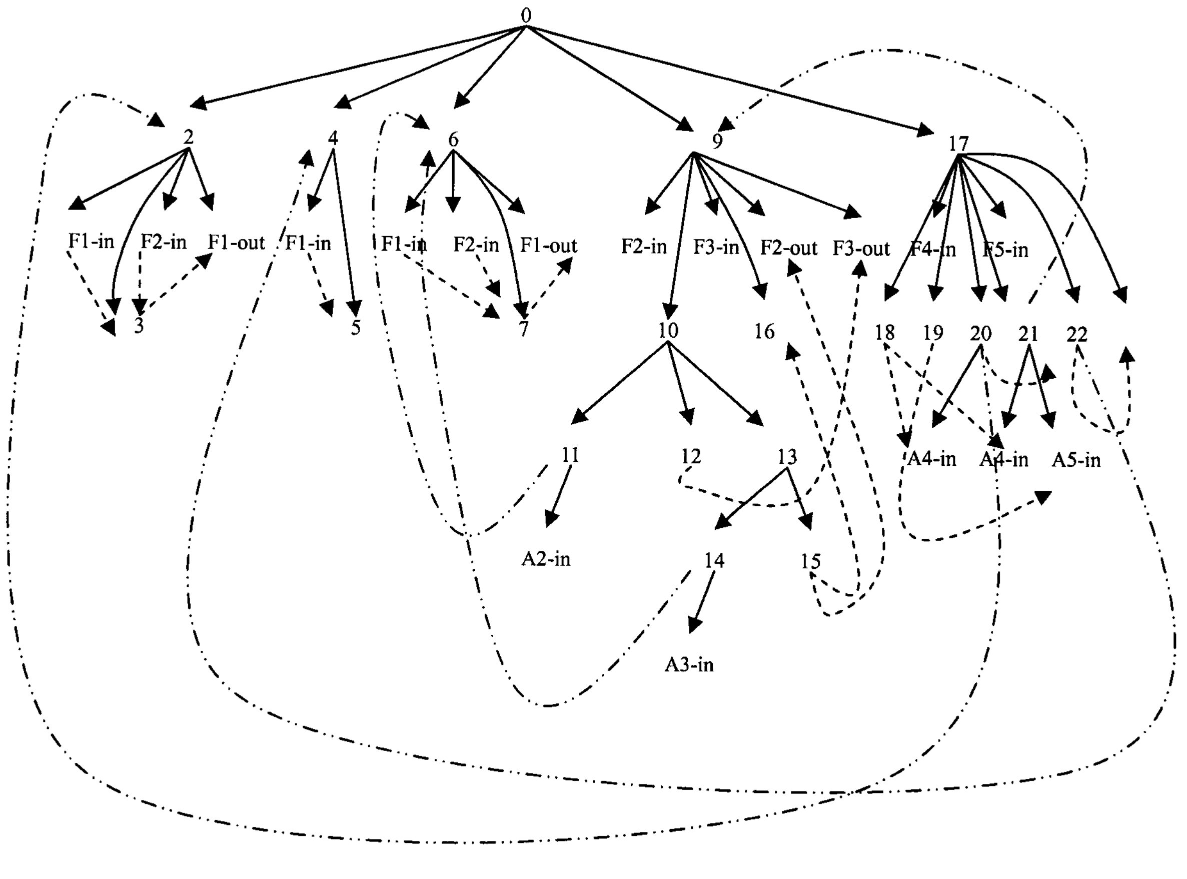


Figure 4. System Dependence Graph of sample program in Figure 1

Using the Horwitz-Reps-Binkley two-pass traversal algorithm [Hor94], the slice with respect to a variable at a point of interest can be computed by including all nodes reachable in the SDG. In detail, during the first pass, the algorithm traverses backward along all edges except parameter-out edges and mark those vertices reached; during the second pass the algorithm traverses backward all vertices marked during the first pass, except call and parameter-in edges and marks reached vertices. Then, the union of the vertices marked during these two passes are included in the slice. Based on this algorithm, the slice with respect to the slicing criterion C (x, 16) for the example in Figure 4 is the following set of nodes: {0, 1, 2, 3, 9, 10, 13, 15, 16, 17, 18, 19, 20, 21}.

Larsen et al. [Lar96] extended the notion of the SDG to compute slices for object-oriented programs, by involving relations between classes in their Class Dependence Graph. The two-pass traversal algorithm is also used.

Livadas et al. [Liv94, Liv95] simplified the computation of the summary edge and other inter-procedural information when constructing the SDG, by considering following situations:

1. When a reference parameter is never modified, no formal-out and actual-out nodes are necessary.
2. When a reference parameter is always modified, that is, the variable is defined in every possible path, formal-out and actual-out nodes are necessary. At the call site, a killing definition that can completely take over previous definitions can be generated for the actual-out node.
3. When a reference parameter is sometimes modified, that is, the variable is not defined in every possible path, or when it is not known how it is used, formal-out

and actual-out nodes are necessary. At the call site, a non-killing definition must be generated for the actual-out node.

4. For value parameters no formal-out and actual-out nodes are necessary.

## 2.2 Dynamic program slicing

A dynamic slice, originally introduced by Korel and Laski [Kor88], is an executable part of the program whose behaviour is identical, for the same program input, to that of the original program with respect to a variable of interest at some execution position. Formally, if a slicing criterion of program P is executed on program input x representing a tuple C = ( x, v(q) ), where v(q) is a variable v at execution position q. A dynamic 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. Furthermore, when the slice is executed on program input x produces an execution trace T'(x) for which there exists the corresponding execution position q' such that the value of v(q) in T(x) equals the value of v(q') in T'(x). Therefore, a dynamic slice P' preserves the value of v for a given program input x.

Obviously, dynamic program slicing is based on the execution trace of the program. An execution trace T(x) is a statement sequence when the program is executed on particular inputs x. For example, the execution trace of the sample program in Figure 1 based on the input: a = 20 and b = 30 is shown in Figure 5.

Notionally, an execution trace is an abstract list (sequence), for instance, T(x) in

Figure 5, T(x)(4) = 20, T(x)(9) = 10. By v(q) we denote variable v at position q, i.e., variable (object) v before execution of node T(x)(q). The notion of an execution position is introduced in this thesis for presentation purpose. Existing dynamic slicing algorithms use the notion of data and control dependences to compute dynamic program slices [Kor98]. The concepts of data and control dependences used by the dynamic slicing algorithm are similar to these defined in static slicing.

*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 5, 23(9) is data dependent on 18(2).

*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 be also extended to actions by using 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 7(14) is control dependent on action 6(13) in the execution trace of Figure 5.

Besides dynamic data and control dependences, another notion of *last definition* LD(v(k)) of variable v(k) in execution trace T(x) has to be considered. By definition, v(p) is v(k)'s LD when action v(p) assign a value to v and v is not modified between position p and k. For example, the last definition of variable "x" at node 16(16) in execution trace of Figure 5 is action 18(2).

```
17(1)     public static void main(String[] args)
18(2)     int a = new Integer(args[0]).intValue();
19(3)     int b = new Integer(args[1]).intValue();
20(4)     Sample s = new Sample(a);
2(5)      public Sample(int x)
3(6)      val = x;
21(7)     int i = 0;
22(8)     while(i < 2)
23(9)     s.min(a, b);
9(10)     public void min(int x, int y)
10(11)    if(x < y)
11(12)    setVal(x)
6(13)     public void setVal(int x)
7(14)     val = x;
12(15)    y = x;
16(16)    Sysetm.out.println("Min is: " + x);
24(17)    a = s.getVal();
4(18)     public int getVal()
5(19)     return val;
25(20)    System.out.println(a);
26(21)    a = a + 10;
27(22)    b = b – 10;
28(23)    i ++;
22(24)    while(i < 2)
23(25)    s.min(a, b);
9(26)     public void min(int x, int y)
10(27)    if(x < y)
13(28)    else
14(29)    setVal(y)
6(30)     public void setVal(int x)
7(31)     val = x;
15(32)    x = y;
16(33)    Sysetm.out.println("Min is: " + x);
24(34)    a = s.getVal();
4(35)     public int getVal()
5(36)     return val;
25(37)    System.out.println(a);
26(38)    a = a + 10;
27(39)    b = b – 10;
28(40)    i ++;
22(41)    while(i < 2)
```

Figure 5. An execution trace of sample program in Figure 1

### 2.2.1 Dynamic Backward Algorithm

The backward algorithm presented by Korel[Kor88] computes a dynamic slice based on dataflow equations. After recording the execution trace during the execution of the program on input x, the dynamic slicing algorithm traces backwards through the execution trace to derive data and control dependencies that are then used for the computation of the dynamic slice.

The execution trace records the executed nodes and the variables defined and used at each executed node. The execution trace is created by embedded monitoring statements, which record the used and defined variables at each node into the original source code. The defined and used variables at each node are identified during the execution by their memory addresses. The instrumentation is performed automatically by a source code parser, which also provides additional information regarding the identification of node types like while loop, selective statements and so forth. Figure 6 shows a part of the instrumented program for the sample program in Figure 1 with the monitoring statements that are used for this implementation of the algorithm.

```
...
     System.out.println( "Node 10\n");                              ←——— Node 10
     System.out.println( "used( " + x'add +";"+y'add+ ")\n");       ←——— Variable x, y are used
10   if( x < y ) {
     System.out.println( "Node 11\n");                              ←——— Node 11
     System.out.println( "used( " +x'add+ ")\n");                   ←——— Variable x is used

11   setVal(x);
...
```

Figure 6. A part of Instrumented Program for the sample program in Figure 1

Based on the execution trace created by the instrumented program shown above, the dynamic slicing algorithm can compute the slice by traversing the execution trace in a backward direction. There are two steps involved in the traversal: one is to find data dependences by finding the last definition of the variable of interest and all variables used in the relevant statements; the other one is to find the control dependences for all statements that were identified earlier as relevant by applying the data dependences.

By combining the identified relevant statements based on data dependences and control dependences, we compute the dynamic program slice for the variables v of interest at execution position p.

### 2.2.2 Dynamic Forward Algorithm

Compared to dynamic backward slicing algorithm, the dynamic forward slicing algorithm [Kor94] does not require the recording of an execution trace and therefore overcomes one of the major weaknesses of the backward algorithm, its space complexity.

The algorithm starts from the entry of the program and computes every variable defined during the program execution for particular inputs. Every time when a statement is executed, the algorithm determines the relevancy of the executed statement for any of variables declared up to the current execution position. The algorithm is illustrated in Table 2, using the sample program in Figure 1.

To improve the comprehension of the illustration, we ignore the function calls and

Table 2. Dynamic forward slicing algorithm

| | Statement | a | b | s | i |
|---|---|---|---|---|---|
| 17(0) | At the beginning | {} | {} | {} | {} |
| 18(1) | int a = new | {18} | {} | {} | {} |
| 19(2) | int b = new | {18} | {19} | {} | {} |
| 20(3) | Sample s = new Sample(a) | {18} | {19} | {20} | {} |
| 21(4) | int i = 0 | {18} | {19} | {20} | {21} |
| 22(5) | while(i < 2) | {18,21,22} | {19,21,22} | {20,21,22} | {21,22} |
| 23(6) | s.min(a, b) | {18,21,22} | {19,21,22} | {20,21,22} | {21,22} |
| 24(7) | a = s.getVal() | {20,21,22,24} | {19,21,22} | {20,21,22} | {21,22} |
| 25(8) | System.out.println(a) | {20,21,22,24} | {19,21,22} | {20,21,22} | {21,22} |
| 26(9) | a = a + 10 | {20,21,22,24,26} | {19,21,22} | {20,21,22} | {21,22} |
| 27(10) | b = b – 10 | {20,21,22,24,26} | {19,21,22,27} | {20,21,22} | {21,22} |
| 28(11) | i ++ | {20,21,22,24,26} | {19,21,22,27} | {20,21,22} | {21,22} |
| 22(12) | while(i < 2) | {20,21,22,24,26,28} | {19,21,22,27,28} | {20,21,22,28} | {21,22,28} |
| 23(13) | s.min(a, b) | {20,21,22,24,26,28} | {19,21,22,27,28} | {20,21,22,28} | {21,22,28} |
| 24(14) | a = s.getVal() | {20,21,22,24,26,28} | {19,21,22,27,28} | {20,21,22,28} | {21,22,28} |
| 25(15) | System.out.println(a) | {20,21,22,24,26,28} | {19,21,22,27,28} | {20,21,22,28} | {21,22,28} |
| 26(16) | a = a + 10 | {20,21,22,24,26,28} | {19,21,22,27,28} | {20,21,22,28} | {21,22,28} |
| 27(17) | b = b – 10 | {20,21,22,24,26,28} | {19,21,22,27,28} | {20,21,22,28} | {21,22,28} |
| 28(18) | i ++ | {20,21,22,24,26,28} | {19,21,22,27,28} | {20,21,22,28} | {21,22,28} |
| 22(19) | while(i < 2) | {20,21,22,24,26,28} | {19,21,22,27,28} | {20} | {21,22,28} |

only show the execution of the "main" function. In above table, we use {} to represent the slice that is a set of statement numbers. Actually, in Korel's algorithm, he did not address how to solve the issue of replacement between the formal parameter and the actual parameter that may happen in the function call, because the algorithm identifies the variable according to the unique memory address not the name. This approach however, is limited to programming languages that facilitate for the retrieval of unique variable memory addresses.

In the algorithm, the slice for the variable defined at current execution position is computed again. That is, current execution position and the slice for all variables used at current execution position will be included. For example, in Table 2, at position 24(7) variable "a" is re-defined, then statement "24" and the slice of used variable "s" – {20,21,22} are included into the new slice for variable "a".

Note that when the complex block [Kor94] like loop or conditional branch is encountered, the algorithm will conservatively include it into the slice and store the slice computed before it into a set of "TopSlice" [Kor94]. After the execution leaves the complex block, the algorithm will determine if it is included eventually based on whether the value of the variable of interest is changed within the complex block. In Table 2, for example, at position 22(19), the slice for variable "s" resumes to {20} same as that before the execution entry the while loop because the value of "s" is not changed within the while loop.

### 2.2.3 Dynamic Dependence Graph

A dynamic variation of the PDG, called the dynamic program dependence graph were introduced by Miller and Choi[Mil88]. Agrawal and Horgan[Agr90] proposed an algorithm to produce more refined slices using the concept of dynamic program dependence graph. Generally, a dynamic slice is determined by computing a static slice in sub-graph of the PDG that is induced by the marked vertices, that is, vertices that are executed. Agrawal and Horgan [Arg90] refined this concept further, by marking the PDG edges, as the corresponding dependences arise during program execution instead of marking vertices to generate more precise slices. Moreover, their later solution is to create a distinct vertex in the dependence graph for each occurrence of a statement in the execution history. The graph that consists of these vertices is called the Dynamic Dependence Graph (DDG).

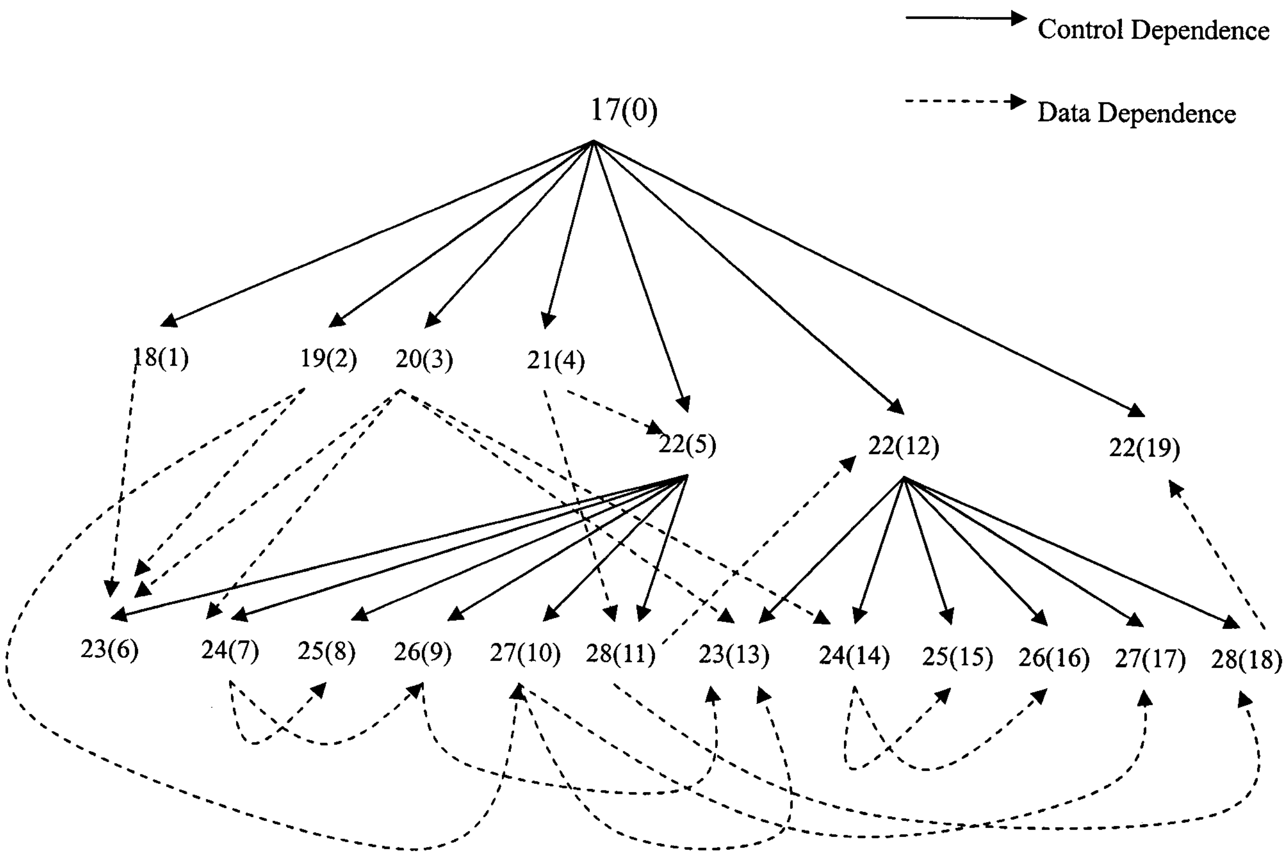


Figure 7 DDG of "main" function of the sample program in Figure 1

A dynamic slicing criterion is identified with a vertex in the DDG, and a dynamic slice is computed by determining all DDG vertices from which the criterion can be reached.

According to the definition of DDG, we can create the DDG corresponding to the "main" function in the sample program in Figure 1 as follows. By backward traversing the DDG from the node of interest, we can get the corresponding dynamic slice easily.

Other algorithms have been proposed since dynamic slicing was originally defined [Kor88]. An inter-procedural slicing algorithm based on dynamic dependence graph have also been proposed by Kamkar et al. [Kam93].

## 2.3 Hybrid slicing

Both static and dynamic slicing have their inevitable drawbacks even though they are widely used in program slicing and keep evolving [Gup97]. In particular imprecision is a problem for the static slicing algorithms, since they are based on all possible executions rather than one specific execution; on the other hand, dynamic slices are more precise but incur high run-time and computation overhead due to the tracing information that is collected during a program's execution and the analysis of these often very large traces. The hybrid slicing algorithm presented in this research attempts to overcome these limitations by combining advantages of dynamic and static slicing. There exist two variations in implementing hybrid slicing: one is to integrate dynamic information from a specific execution into a static slice analysis to produce more precise slices; the other is to embed static slicing into dynamic slicing to avoid recording some of the program constructs that cause these large traces.

### 2.3.1 Breakpoint History

Rajiv Gupta et al. [Gup97] proposed a hybrid slicing algorithm in 1997, which made static slicing more precise by using breakpoint information and the dynamic call graph. It is actually an approach to integrate dynamic information into static program slicing. In this algorithm, the breakpoints and call/return points are used as reference points, to divide the execution path into intervals. The integration of dynamic information into a static analysis can improve the accuracy in estimating which control flow path is potentially taken by the program.

This approach assumes that there is at least one breakpoint set in the program during execution, and the overall slice is split into sub-slices corresponding to relevant statements that could have been executed between every successive pair of breakpoints encountered so far. The following are some definitions used in their algorithm to compute hybrid slices.

(1) The *breakpoint history* of a program execution is of the form

$$BH = <(b0,\ \Phi), (b1, N1 = B1-\{b1\}), \ldots (bm, Nm = Bm-\{bm\})>,$$

Where b0 is the start node of the program; bm is the latest breakpoint encountered; Bi is the set of breakpoints that were active during the execution interval from breakpoint bi-1 to bi; and Ni is the set of breakpoints that were active but not encountered during the execution interval from bi-1 to bi.

(2) A *slicing criterion* is of the form SC = (V, b), where V is the set of variables whose values are of interest at breakpoint b.

(3) For a given breakpoint history, BH = <(b0, $\phi$), (b1, N1), … (bm, Nm)>, the hybrid slice with respect to a slicing criterion SC = (V, bm) is defined as follows:

$$\text{HSLICE(b0, bm)} = \bigcup_{i=1}^{m} \text{HSLICE(bi-1, bi)},$$

where HSLICE(bi-1, bi) contains those statements that were possibly executed after the breakpoint bi-1 and prior to breakpoint bi and where their execution, directly or indirectly, influenced the computation of the value of some variable in V at bm. The statements that influence the computation of a variable are computed by taking the transitive closure over both data and control dependences.

By involving call/return information, the algorithm is extended to deal with inter-procedural slicing. Other relative definitions are introduced as follows:

(1) The *call history* of a program execution is of the form CH = <CR0, CR1, …, CRm>, where Cri is either a procedure call or a return from a procedure, and CR0 is assumed to be a call to start execution of the main program.

(2) For a calling history, CH = <CR0, Cr1, …, CRm>, the overall *hybrid slice* with respect to a slicing criterion SC = (V, CRm) is defined as flows:

$$\text{HSLICE(CRm)} = \bigcup_{i=1}^{m} \text{HSSLICE(CRi-1, CRi)},$$

where subslice HSSLICE(CRi-1, CRi) contains those statements that were possibly executed after CRi-1 and before CRi, and their execution, directly or indirectly, influenced the computation of the value of some variable in V at CRm.

(3) Given a call/return history, CH = <CR0, CR1, … , CRm>, a path from CR0 to CRm is feasible if and only if path P is composed of the following subpaths:

P = PATH(CR0; CR1).PATH(CR1;CR2)....PATH(CRm-1; CRm).

(4) The *combined history* of a program execution is of the form H = <H0, H1, ..., Hm>, where Hi is one of the following: Breakpoint: (bi, Ni); Procedure Call: CALL{P(caller) $\rightarrow$ P(callee) at s}; or Procedure Return: RET[P(callee) $\rightarrow$ P(caller) at s].

(5) For a given combined history, H = <H0, H1, ... , Hm>, the overall hybrid slice with respect to a slicing criterion SC = (V, CRm) is defined as follows:

$$\text{HSLICE(Hm)} = \bigcup_{i=0}^{m-1} \text{HSSLICE(Hi, Hi+1)},$$

where subslice HSSLICE(Hi, Hi+1) contains those statements that were possibly executed after Hi and before Hi+1, and their execution, directly to indirectly, influenced the computation of the value of some variable in V at CRm.

The limitations of this approach are that it only supports slice computation for structured programs and the computed slices may not be executable [Ril01]. In addition, another shortcoming is that users are required to interact with the program to determine when and where the breakpoints are set.

### 2.3.2 Call-Mark Slicing

Another approach of hybrid program slicing, called Call-Mark Slicing, can also be seen as a refined static slicing approach by including dynamic information, as proposed by Akira Nishimatsu et al. [Nis99]. Their approach is based on the PDG and computes the slice by only analyzing valid nodes according to execution dependences.

The approach introduces the Concept of Execution Dependence (CED), in which, a statement s1 is execution dependent on another statement s2, if execution of s2 is necessity of execution of s1. CED(s), corresponds to a set of caller statements with execution dependence, that can be defined as follows:

CED(s) $\equiv$ {t | t is a function/procedure call statement and s is executionally dependent on t}

There are three steps to compute the call-mark slice:

1) Construct PDGs similarly to other static slicing algorithm and create CED(s) for each statement;

2) Execute the program based on a particular input, then mark call statements and insert them into a set called CM ;

3) Traverse backwards from the node of interest without considering a node n if CED(n) $\cap$ CM = {}.

This approach is suitable for analyzing procedural languages like C, but it does not apply for object-oriented languages, because it is very difficult to derive the CED(s) for object-oriented language constructs, in particular for inheritance and implicit function calls.

### 2.3.3 Conditioned Slicing

The Conditioned Slicing [Can98] is a kind of hybrid slicing that combines static and dynamic slicing by augmenting the static slicing criterion with a condition. It is formally defined as a slice that is constructed with respect to a tuple, (V, n, $\pi$), where V is a set of variables, n is a point in the program and $\pi$ is some condition. Firstly, the program is simplified with respect to input condition. Then the slice is computed based on the reduced program using static slicing. For example, in Figure 8, a reduced program based on the sample program in Figure 1 that satisfy the condition $\pi$ = (a =1 AND b = 100) is shown. Then a slice with respect to a slicing criterion C = (a, 23) is computed {0,1,2,3,4,5,6,7,9,10,11,15,16,17,18,19,20,21,22,24,25,26}.

The process of reducing the program is based on the identification of infeasible paths, which can be computed by the symbol executor [Kin76, Cow88] and be automatically evaluated by a theorem prover [Boy79].

An automatic conditioned slicer is implemented by Danicic et al. [Dan00] in their ConSIT system. A backward conditioning was also proposed to extend the conditioned slicing based only on forward conditioning. The conditioned slicing based on symbolic execution however has limitations, with respect to the input conditions which can be handled by the symbolic execution and the language constructs supported. Other limitations include restrictions with respect to language constructs and program languages that are supported by both, the symbolic execution environment and the theorem prover.

```
public class Sample{
   private int val;
   public Sample(int x){
      val = x;
   }
   public int getVal(){
      return val;
   }
   public void setVal(int x){
      val = x;
      return;
   }
   public void min(int x, int y){
      if(x < y){
         setVal(x);
         y= x;
      }
      System.out.println("Min is: " + x);
   }
   public static void main(String[] args){
      int a = new Integer(args[0]).intValue();
      int b = new Integer(args[1]).intValue();
      Sample s = new Sample(a);
      int i = 0;
      while(i < 2){
         s.min(a, b);
         a = s.getVal();
         System.out.println(a);
         a = a + 10;
         b = b - 10;
         i++;
      }
   }
}
```

Figure 8. Reduced sample program with respect to a condition

## 2.4 Applications of program slicing

Program slicing was originally introduced as a technique that can be used to localize program errors [Wei82]. In the last couple of years, the application domain of program slicing has been extended into various areas of software maintenance, like program differencing, program integration, software maintenance, program testing, debugging, reverse engineering and program comprehension.

- **Program Differencing**

In the process of program development, only newly changed components are usually interesting to be tested. Thus, program differencing [Hor89], which identifies the new change syntactically and semantically is useful. The key issue in program differencing is to partition the components of the old and new version in a way that two components are in the same partition only if they have equivalent behaviours [Tip95]. Program slicing can be applied in program differencing, by computing and comparing slices and their behaviours, before and after performing a modification.

- **Program Integration**

Program Integration [Hor90, Hor89] is a process to merge two programs A and B that both resulted from modification to the base version of the program. Program integration relies on program slicing to produce slices with respect to the affected points and compare them to determine if A and B can be integrated with each other.

- **Software Maintenance**

Ripple Effect Analysis (REA) determines whether a change at some place in a program will affect the behaviour of other parts of the program. Ripple effect analysis plays a very important part of software maintenance. Both backward and forward program slicing can be applied within REA [Wan97].

- **Program Testing**

Data flow testing can be applied to check if all def-use pairs (pairs of variables defined and used ) occur in a successful test case. In this process, program slicing is used to construct test-cases that test all def-use pairs [Kam93]. Regression testing reuses existing test cases to test only these parts of the program that are affected by a modification of a previously tested program. Program slicing can be used to determine the program parts that are affected by the change [Gup92]. In addition, it can also be used in incremental program testing [Bat93].

- **Debugging**

Debugging is the process of finding bugs in a program that lead to an unexpected program behaviour. The backward slice can be used to extract those statements that affect the variable at the point where the wrong value is produced. Moreover, static slicing can be used to detect the "dead" code that cannot affect any output of the program [Ber85]. Agrawal [Agr92] discussed how dynamic and static slicing could be utilized for semi-automated debugging of programs.

- **Reverse Engineering and Program Comprehension**

Reverse Engineering is a promising approach to combat legacy system problems [Hau99]. Program comprehension, however, is the most important and hardest part of the process of reverse engineering. Program slicing, abstracting out of source code the design decision by reducing the amount information to be analyzed and therefore improving the comprehension process. Furthermore, program slicing can be used to provide the maintainer and reverse engineer with additional insights and abstractions of the legacy system under investigation.

# 3 Contributions

The CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) project aims to provide and introduce novel approaches to software comprehension. An important part of CONCEPT project is to support the comprehension process, by utilizing various source code analysis techniques, namely program slicing and different visualization techniques. The hybrid algorithm presented in this research simplifies the program to be analyzed and provides the basic input for further analysis and visualization of various program aspects.

## 3.1 Motivation

Though there have been many program slicing algorithms presented in the literature, all of these algorithms have limitations with respect to their program language support, slicing precision, parsing requriments, etc, These limitations make the integration and use of these algorithms difficult. The introduction of our hybrid slicing algorithm is based on the following motivations: combining advantages of static and dynamic slicing, computing user specified executable hybrid slices, analyzing Java programs and implementing the hybrid slicing algorithm.

- **Combining Advantages of Static and Dynamic Slicing**

Both the static and dynamic program slicing approaches have their respective advantages and disadvantages even though they are effective techniques used in various application domains ranging from program debugging to software reverse engineering. Static program slicing that has to consider all possible inputs computes often program slices that are imprecise. Especially when applied for objected-oriented programming languages these limitations of static slicing become more obvious. Based on the static nature of the algorithm and the necessity to consider all possible executions, the algorithms have to make rather conservative assumptions with respect to dynamic language constructs. For instance, polymorphism, an important feature that is often used in object-oriented programs, may involve undetermined path at a call site, which will be determined only at run time. Thus, static program slicing that uses only static information cannot determine which path should be taken. Conservatively, all paths have to be considered. The slice, therefore, is finally imprecise and often larger when necessary.

Dynamic program slicing has many advantages, compared to static program slicing, in particular when object-oriented programs are analyzed. Advantages can be found in the handling of dynamic language constructs. These dynamic aspects are resolved by utilizing run-time information and therefore only considering the statements that were executed. However, dynamic program slicing has also several disadvantages. The major limitation of dynamic slicing is its high run-time overhead due to recording of the program execution and/or run-time analysis of every executed statement [Ril01]. It is the time and space complex (unbound space and time-complexity) caused by recording execution traces, especially in situations when the program involves large loops. For example, in the case that a program has a "for" loop that is traversed for **n** times and there are **m** statements within the loop, the total number of statements that are stored and analyzed is **n x m.** If both **n** and **m** are amplified to very big numbers, the recording of execution trace and analyzing of the same could be a nightmare. In addition, the execution of many statements that are recorded and analyzed may actually not be relevant to the slice criterion. Since both static and dynamic program slicing have their inevitable disadvantages, hybrid program slicing tries to overcome some of these limitations, by combining the advantages of both algorithm, to derive an algorithm that computes slices that are more precise than traditional static slices and at the same time have a lower overhead and cost associated than the traditional dynamic slicing algorithms.

- **User specified Executable Hybrid Slice**

Existing approaches to hybrid program slicing either integrate dynamic information into static program slicing or involve static computation in dynamic program slicing. Gupta et al. [Gup97] proposed an approach that uses dynamic information to make the static slice more precise. The dynamic information that the approach uses is breakpoint information and the dynamic call graph, which is readily available during debugging. Thus, the slicing criterion is no longer only the pair of variable of interest and point. The breakpoint or call/return point relative to an execution point is also included into the criterion. For example, if a data dependency of the variable of interest is found, it is not included into the slice immediately as static program slicing does. The approach must ensure that the path on which the dependency is identified is feasible according to the dynamic information. This approach produces the hybrid slice more precise than the static slice and less costly than the dynamic slice. However, it only supports structured programs [Rilling01] and the computed slice may not executable because it divides the program into several intervals based on the breakpoint that might lead to the exclusion of some statements that are relevant to make the slice executable. Another shortcoming [Ril01] of this approach is that it requires the user to interact with the program to determine when to set the breakpoint. Call-mark slicing [Nis99] is another approach to hybrid program slicing, in which there is no any user involvement. Other approaches to hybrid program slicing have been proposed [Cho91, Due92, Kam93, Net94]. All of these approaches embed static computation in dynamic slicing, but they do not allow the user to selectively adopt the algorithm to the specific needs. One major goal of the presented hybrid slicing algorithm has to be to compute a precise slice using dynamic slicing without the necessity of recording the execution trace that generates the space and time complexity. In our approach we provide the user with the ability to determine what parts of the program should be analyzed statically and which parts are analyzed dynamically. Leaving the user with the ability to select the required slice precision and computation complexity based on the particular needs.

- **A Hybrid Slicing Algorithm for Java Program**

Java is a portable and platform independent object-oriented language, which has been used in Web based programming as well as in many mission critical applications. Maintenance to Java programs is getting more and more important and urgent. However, it is very difficult to create such a program slicing algorithm that can successfully analyze any languages and so far it does not exist because every programming language has its unique syntactical and semantic features. For example, Java's mechanism of exception handling is different from that of C++, although they are both object-oriented languages. Existing approaches to hybrid program slicing including those mentioned in the literature review support procedural languages. For example, Gupta et al's approach [Gup97] is implemented to analyze the program written in ANSI C, and the approach proposed by Schoening and Ducase [Sch95] only supports Prolog. In addition, as a matter of fact, so far there is no effective hybrid slicing algorithm particularly for Java and object-oriented programs.

- **An Implemented Hybrid Slicing Algorithm**

It is difficult [Tom95] to compare program slicing tools because of several reasons: 1) they are generally very language/dialect specific; 2) Even if the tools are suppose to compute slices for the same program language, they often only handle a subset of these languages; and 3) constructing test programs of reasonable size is difficult. Anyway, the bottom line of evaluating program slicing tools is that they must be implemented and executable. However, many program slicing algorithms have not been implemented or are only partially implemented to analyze some simple sample programs. They are often limited in their capability of handling real world programs. For hybrid program slicing, this situation is even more serious. For example, Gupta et al's algorithm [Gup97] is implemented to analyze programs with less than 700 lines of source code. In their article they do not address the issue of scalability and performance of their algorithm for large program. In order to evaluate performance of slicing tools between hybrid slicing and others, the hybrid slicing algorithm must be implemented and has to be able to analyze real software system. Therefore, the implementation have to be considered carefully on how to extract useful information from the source code, and issues of information storage as well as retrieval have to be addressed carefully.

## 3.2 The Notion of Removable Blocks

The notion of removable blocks was first introduced by Korel [Kor92] to represent the smallest component of program text that can be removed during slice computation without violating the syntactical correctness of the program. Every statement including assignment statements, input and output statements and selective statements can be a block. Moreover, a block can be a complex block or a simple one. If a block has other blocks nested in it, it is a complex block. Otherwise, it is a simple block. For example, a while loop that could contain some assignment statements is a complex block and every assignment inside the loop block is a simple block. In addition, a block can be non-removable or removable, which means a block can be either included into the slice or not. Furthermore, if any inner block is to be considered as a non-removable and be included, all the outer blocks encompassing the inner block are included automatically. Likewise, if any outer block is to be included, automatically all the inner non-removable blocks are included.

The advantages of the notion of a block are that it can represent not only control dependences but also syntax dependences. That is, some statements in the program that seem irrelevant to the point of interest based on control or data dependences, in order to make the slice executable, can be considered as non-removable and included into the slice. For example, the statement like "*import java.util.*;*" is regarded as a non-removable block and included into the slice to make the slice compilable and executable.

Our hybrid slicing algorithm uses the same notion of removable/non-removable blocks to ensure the computed slice executable.

## 3.3 Static Program Slicing Algorithm

Our static program slicing algorithm combines the syntax dependence and the notion of removable blocks. By traversing the syntax trees, the algorithm identifies the data dependences between the relevant block and the variable of interest, relying on a collection of data flow equations. On the other hand, the algorithm uses the notion of removable blocks to compute the control dependence and ensure therefore that the computed slice is executable. The details of the static slice computation are shown in

Figure 9.

```
Input:   a slicing criterion C=(v, n)
Output:  a static program slice for C

S:   Set of blocks in the slice
MB:  Set of marked blocks
v:   Interesting variable
n:   Interesting block

1.  Initialize S={ }, MB={ }.
2.  MB = MB U {n}
3.  do
4.    Find last definition m of variable v at n
5.    if(m ∉ S and m ∉ MB)
6.      MB = MB U {m}
7.    end-if
8.    Let block B be n's immediate outer block
9.    if(B ∉ S and B ∉ MB)
10.     MB = MB U {B}
11.   end-if
12.   if(block C is n's inner block and C is non-removable and C ∉ S)
13.     S = S U {C}
14.   end-if
15.   S = S U {n}
16.   MB = MB - {n}
17.   Let n be the first element of MB
18.   Let v be used variables at n
19. while MB = { }

20. procedure Find last definition
21. Let block m is the precedent for n in syntax tree
22. while m is not the top of the syntax tree
23.   if (v is not in variables defined in m)
24.     n = m;
25.     Let block m is the precedent for n in syntax tree;
26.   else
27.     break;
28.   end-if
29. end-while
30. end Find last definition
```

Figure 9. Static slicing algorithm

The algorithm is straightforward. Step 4 to step 7 calculates the last definition of the current variable of interest. Then the found block on which the current block of interest is data dependent is put in the set of marked blocks, *MB,* for further analysis. Note that there may be multiple last definitions found because the static slicing algorithm cannot completely know which path will be taken when the case like conditional branch or polymorphism is encountered, but conservatively has to consider all possibilities. All of blocks found are stored in the set *MB* no matter how many blocks are found. After solving the data dependence, step 8 to step 11 addresses the issue of control dependence. This is actually solved by including the outer block of the current block of interest into the set *MB*. Nested blocks can express the control dependence between blocks. Step 12 to step 14 includes all non-removable blocks nested in the current block into the slice set to ensure that the computed slice is executable. Once the analysis for a variable at a block of interest has finished, this block is included into the slice and removed from the set of marked blocks. The algorithm will repeat the same process to analyze every variable used in the block in the set *MB* until the set is empty.

The most important and complicated part of the algorithm is the procedure of searching for the last definition of the variable of interest. The search involves navigating the syntax trees, starting from the point of interest. The syntax tree consists of all removable blocks that are nodes and control dependences that are edges. Each tree represents a free function in the program and call edges are used to connect trees between the caller and the callee. For example, the syntax trees of the sample program in Figure 1 is shown below in Figure 10.

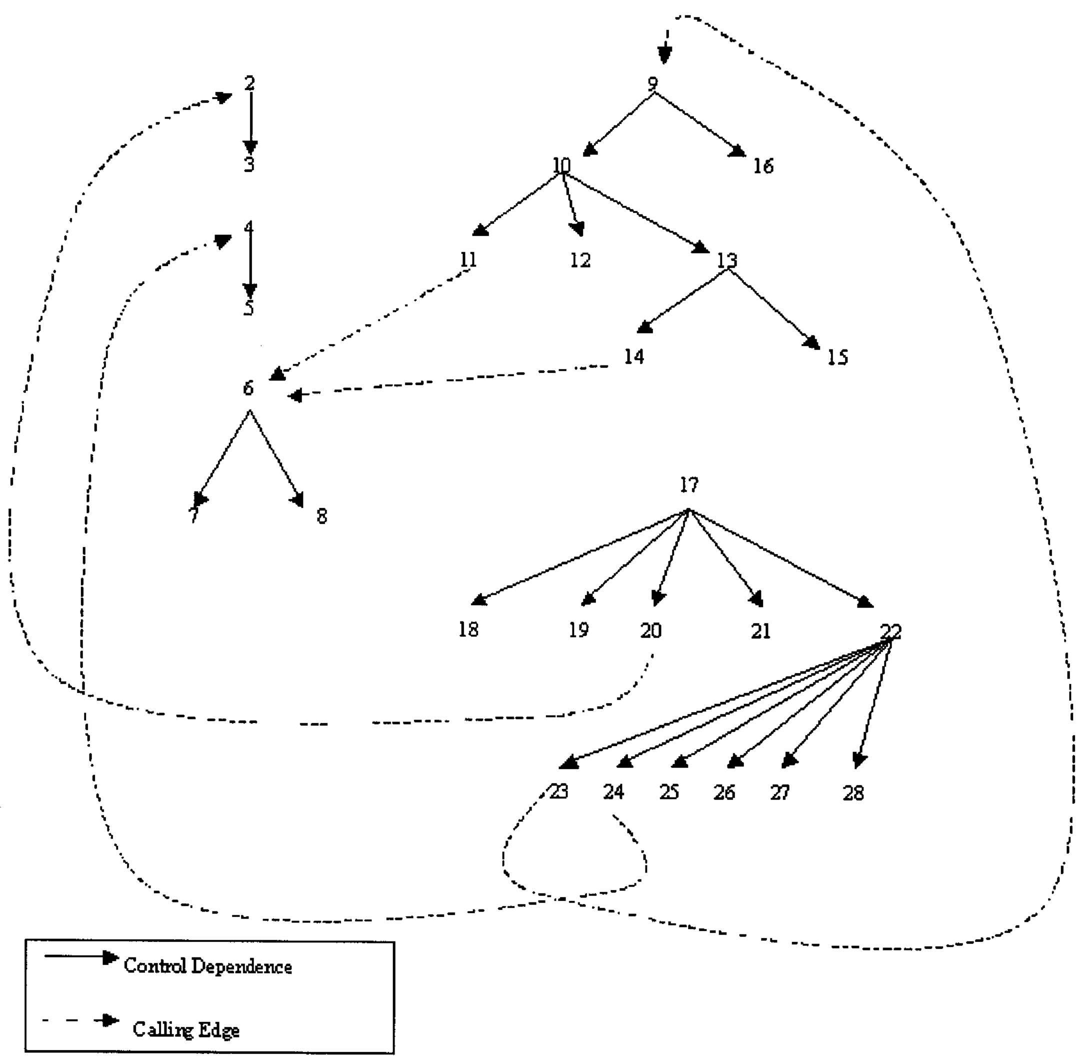


Figure 10. Syntax trees of sample program in Figure 1

Assuming a slicing criterion C = (*a*, 25), the search for the last definition of variable *a* in the block 25 starts from the node 25 then moves "left & up" towards the root block. In addition, the navigation jumps to the relevant syntax tree when a function-call is encountered. In Figure 9, the navigation jumps to node 4 when it arrives node 24 because there is a function call in block 24. Every time the navigation jumps to a syntax tree, it always follows the "left & up" traversing principle that it starts from the last node and moves "left & up" towards the root node. For example, after the navigation jumps to node 4, it goes to node 5 immediately and starts the traverse within the new syntax tree where node 4 is the root.

Then the navigation returns back to the call site and continues the search. In above case, the navigation goes back to node 24 after it finishes visiting node 4. Usually, if the last definition is found, the navigation stops immediately. For example, when the last definition of variable *a* is found in node 24, the navigation stops and the algorithm begins to handle the control dependence. However, there are still some special cases that have to be addressed as follows: conditional branch, loop, inter-procedural navigation, inheritance, exception handling and polymorphism.

### 3.3.1 Conditional Branch

The static slicing algorithm has to consider all possibilities when it handles conditional branches, because the algorithm does not know which path the program takes. There are two kinds of conditional branch in a Java program: if … else and switch … case.

#### 3.3.1.1 IF … ELSE

"If … else" is a conditional statement that is often used in the program. There are four kinds of cases that the static slicing algorithm has to handle: traverse from the true case, traverse from the false case, the last definition found in the true case and the last definition found in the false case.

**Traverse from the true case**

In this case, the static slicing algorithm just traverses "left & up" over the syntax tree, like the normal case. The traverse from node 12 in Figure 9 is such an example.

**Traverse from the false case**

When traverse starts from the false case to search for the last definition of the variable of interest, the true case of the same conditional statement will be ignored. That is, the traverse will skip the nodes within the true case to get the "if" statement directly. The reason is that simultaneous execution of the true case and the false case never happens. The algorithm has to find and visit the node that corresponds to the "if" statement after the traverse visits the "else" node. Then the traverse continues from the "if" node. For instance, the traverse starting from node 15 will skip node 11 and 12.

**The last definition found in the true case**

If the last definition of the variable of interest is found in the true case, the algorithm must go to the false case to perform another search because the static slicing algorithm cannot know which case will be executed. If the last definition is also found in the false case, the search will stop. Otherwise, the search will continue beyond the "if … else" boundary. In Figure 9, for example, if the last definition of variable y is found in node 12, the search will continue to node 15 rather than stop. In case there is nothing found in the relevant false case ranging from node 15 to 13, the algorithm will go on to search "left & up" from node 10.

**The last definition found in the false case**

Similar to the previous case, if the last definition is found in the false case, the algorithm has to do another search in the true case. The search must continue if nothing is found in the true case. The example mentioned above can be also used here.

#### 3.3.1.2 SWITCH ... CASE

Although the "switch ... case" is similar to the "if ... else", it is handled by the algorithm differently. There are still two types of cases to be considered: traverse from a case block and the last definition found in a case block.

**Traverse from a case block**

Normally, if the traversal starts from a case block, the algorithm will ignore the other case blocks and go to the "switch" statement directly.

**The last definition found in a case block**

If the last definition of the variable of interest is found in a case block, the algorithm must continue to search in every other case block. The traversal has to go on outside the "switch" block if nothing is found in any one of other case blocks.

### 3.3.2 Loop

There are three kinds of loops in Java: for, while and do loop. The static slicing algorithm, deals with these loops in a similar fashion.

#### 3.3.2.1 Traverse from the loop node

Here the loop node corresponds to "for" statement, "while" statement or "do" statement that represents the beginning of a loop. In this case, the traversal cannot go "left & up" as usual because the statement could be affected by the statements after it. Hence the traversal has to start from the last statement of the loop block. If the last definition is found within the loop block, the position will be recorded and the traversal within the loop block will stop, then the traversal continues from the loop statement. For example, in Figure 9, the traverse starting from node 22 has to visit the nodes 28 to 23 before it continues traversing to node 21.

#### 3.3.2.2 Traverse from a node within the loop

Since the static slicing algorithm cannot know how many times the loop iterates, the algorithm assumes that there exists at least one iteration. Therefore, the last definition of the variable used in the statement within the loop block can be found before or after the statement, because the control flow is based on at least one loop iteration. Thus the static slicing algorithm must traverse the nodes both before and after the starting node. In Figure 9, for example, the traverse starting from node 25 must go to node 28 after node 22 is encountered.

### 3.3.3 Inter-procedural Navigation

In addition to the special cases mentioned above, jump calls between syntax trees involve an inter-procedural navigation that needs to be treated specially. Although the system dependency graph (SDG) [Hor89] proposes a way to handle inter-procedural program slicing, resolving the issues in inter-procedural navigation to compute a correct and precise slice is still very difficult, especially for object-oriented programs. There are too many possibilities that have to be considered because Java program consists of a large number of function calls that could be found everywhere including assignment expressions and parameter lists. In this paper the inter-procedural navigation is classified into two types: outside-in and inside-out.

#### 3.3.3.1 Outside-in Inter-procedural Navigation

The traveral that goes from the call site of a function to the body of that function is called outside-in inter-procedural navigation. Sometimes a variable's value could be changed in a function along with the change of the variable name because of the exchange between the actual parameter and the formal parameter. In static slicing algorithm, the variable is recognized according to its name that could be changed when the outside-in inter-procedural navigation happens. For example, variable *a* is changed to *x* when the navigation jumps from node 22 to node 9. The algorithm replaces all formal parameters with their corresponding actual parameters when the jump happens at the call site. Actually, in Java, only object parameters can be changed permanently. Therefore, replacement only happens when the parameter is an object.

#### 3.3.3.2 Inside-out Inter-procedural Navigation

The traversal that goes from the body of a function to the call site of that function is called inside-out inter-procedural navigation. The case of inside-out inter-procedural navigation is similar to that of outside-in. In this case the algorithm replaces all actual parameters with their respectively corresponding formal parameters when the navigation jumps to the call site. In Figure 9, variable *x* is replaced by *a* when the navigation jumps from node 9 to node 20.

### 3.3.4 Inheritance

Inheritance between classes can be recorded as a relationship in the block. If the current class is included into the slice, the algorithm will find its base class based on the recorded relationship and include the base class into the slice.

### 3.3.5 Exception Handling

Since the static slicing algorithm does not know when and which exception happens and will be caught, we have to conservatively include all catch statements into the slice in order to make the slice executable.

### 3.3.6 Polymorphism / Dynamic Binding

Polymorphism and dynamic binding provide much of the power of object-oriented programming but at the same time, they represent also a major challenge for static slicing algorithms. Traditionally, the static slicing algorithm has to handle them conservatively. That is, it has to consider all possibilities. Our static slicing algorithm tries to solve this issue by checking the object's type to determine which path will be taken. However, the issue can only be solved partially because the algorithm still cannot identify what the type of the object is when it is represented by an element of an array. In this case, the algorithm will take the traditional method to consider all possibilities.

## 3.4 Dynamic Slicing Algorithm

Our dynamic slicing algorithm looks similar to the static slicing algorithm except that static blocks are replaced by dynamic actions combining the block with the execution position, and that the navigation for finding the last definition of the variable of interest is no longer based on the syntax tree but the execution trace. Details of the algorithm are shown in Figure 10.

```
Input:    a slicing criterion C=(v, n(q))
Output:   a static program slice for C

S:      Set of blocks in the slice
MB:     Set of marked blocks
v:      Interesting variable
n(q):   Interesting block n at position q

1.  Initialize S={ }, MB={ }.
2.  MB = MB U {n}
3.  do
4.    Find last definition m(p) of variable v at n(q)
5.    if(m(p) ∉ S and m(p) ∉ MB)
6.      MB = MB U {m(p)}
7.    end-if
8.    Let block B(p) be n(q)'s immediate outer block
9.    if(B(p) ∉ S and B(p) ∉ MB)
10.     MB = MB U {B(p)}
11.   end-if
12.   if(block C(p) is n's inner block and C(p) is non-removable and C(p) ∉ S)
13.     S = S U {C(p)}
14.   end-if
15.   S = S U {n(q)}
16.   MB = MB - {n(q)}
17.   Let n(q) be the first element of MB
18.   Let v be used variables at n(q)
19. while MB = { }

20. procedure Find last definition
21. Let block m(p) is the precedent for n(q) in the execution trace
22. while m(p) is not the top of the execution trace
23.   if (v is not in variables defined in m(p))
24.     n(q) = m(p);
25.     Let block m(p) is the precedent for n(q) in the execution trace
26.   else
27.     break;
28.   end-if
29. end-while
30. end Find last definition
```

Figure 11. Dynamic slicing algorithm

In the first step of the algorithm, the program is executed and its execution trace is recorded up to execution position $q$ of interest. Then the search for the last definition of the variable of interest starts from the execution position $q$ and moves backward through the execution trace. After the last definition is found, it is included into the set of marked actions *MB*, referring to lines 5-7 in the algorithm. The next step is to handle control dependences that correspond to actions where the block is the control dependency of current block of interest and the execution position is closest to the execution position of interest. For example, in the execution trace shown in Figure 5, control dependence of action 25(37) is action 22(24), not 22(8) or 22(41). Furthermore, the identified control dependency is stored in the set *MB* for further analysis. An executable slice is computed by including all non-removable blocks within the currently analyzed block. In the next step, the block is removed from the set *MB* and included into the slice. The algorithm continues with the next iteration by analyzing the first element of *MB* and the variables used in this action. The loop terminates when the set *MB* is empty, which means that the analysis of all marked blocks is completed. Of course, some duplicates that have same blocks but different execution positions are merged into one unique block in the final result of the slice. For example, there may be actions 22(8), 22(24) and 22(41) in the slice, which are merged into one block 22 finally.

The most challenging part of the dynamic slicing algorithm is the process of searching for the last definition, even though it is much easier than that of static slicing algorithm. The process involves a navigation based on the execution trace that automatically resolves branches caused by conditional statements or polymorphism. However, the algorithm has to consider the change of the variable when inter-procedural navigation is encountered, since the variables are still identified by their variable name. The reason is because gaining the memory address of a variable in a Java program is much more difficult than in a C++ program. Like static slicing algorithms, dynamic slicing algorithms also take two kinds of inter-procedural navigation into account: outside-in and inside-out.

**Outside-in Inter-procedural Navigation**

When the navigation moves follows a function call to another function, and if the variable of interest is an object and passed as an actual parameter, the variable must be replaced by its corresponding formal parameter (also referred to as a reflection). For example, in Figure 5, variable a will be replaced by variable x corresponding to action 24(17) and action 16(16).

**Inside-out Inter-procedural Navigation**

If the current variable of interest is a formal parameter of a function, and the navigation is moving outside, the variable should be replaced by its corresponding formal parameter at the nearest call site. In Figure 5, for example, variable x should be replaced by variable a when the navigation from action 9(26) to action 23(25).

## 3.5 Hybrid Slicing Algorithm

The general hybrid slicing algorithm presented in this research attempts to utilize advantages of both static and dynamic slicing algorithms. We design and implement two types of hybrid slicing algorithm [Ril01]: basic hybrid slicing algorithm and criterion based hybrid slicing algorithm.

### 3.5.1 Basic Hybrid Slicing Algorithm

The basic hybrid slicing algorithm is just a simple combination of static and dynamic slicing algorithms. First of all, the algorithm uses the static slicing algorithm to produce an executable slice that then is analyzed by the dynamic slicing algorithm as a reduced source code. Figure 12 illustrates how the basic hybrid slicing algorithm works.

According to Figure 12, the source code is first passed through the static slicing algorithm to become an executable static slice. This process can reduce the number of statements in the source code significantly (depending on the slicing criterion and the cohesiveness of the program). However, the slice is not as precise as a dynamic slice, because the static slicing algorithm must consider all possible paths when it handles conditional branches, polymorphism and dynamic bindings. After the computation of the static slice, a dynamic slice for the same slicing criterion, except this time for a specific input can be applied on top of the static slice to further refine the slice. Moreover, based on the reduced source code rather than original source code, the dynamic slicing algorithm has less of an overhead for recording the execution trace, because only the statements that are potentially relevant to the slicing criterion are recorded. Finally the hybrid slice is generated.

### 3.5.2 Criterion Based Hybrid Slicing Algorithm

The criterion based hybrid slicing algorithm is an enhancement of the previous presented general hybrid slicing algorithm that allows the user to combine static and dynamic analysis on an *as-needed* basis. Our hybrid slicing algorithm is based on the hybrid slicing framework proposed in [Ril01]. The algorithm is illustrated in Figure 13.

For the criterion based hybrid slicing algorithm, static analysis is used to reduce the overhead required by the trace and run-time evaluations associated with the dynamic slicing algorithm.

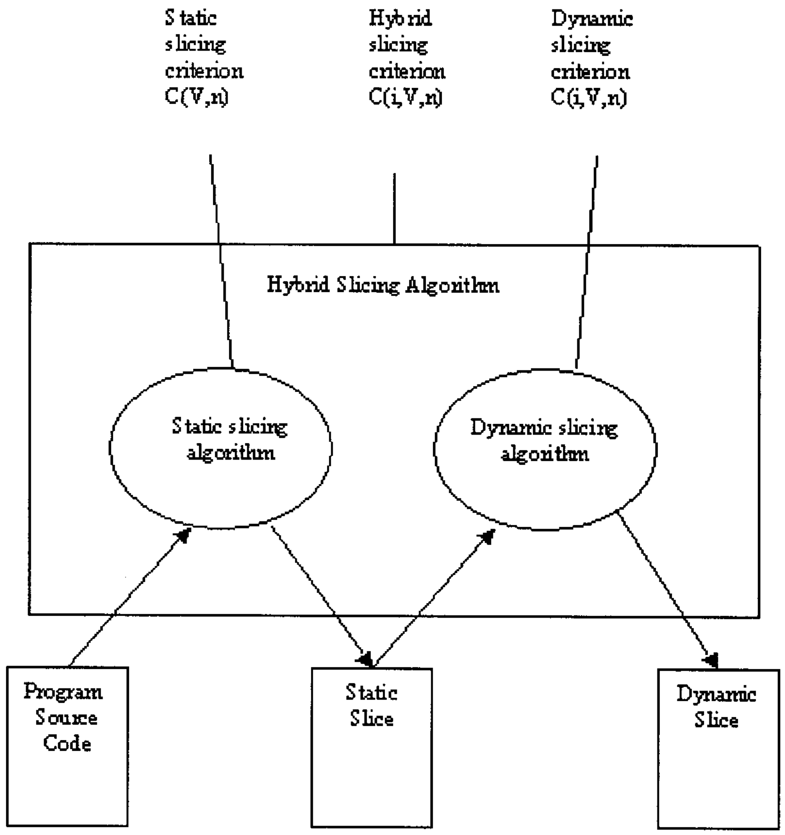

Static slicing criterion C(V,n)
Hybrid slicing criterion C(i,V,n)
Dynamic slicing criterion C(i,V,n)
Hybrid Slicing Algorithm
Static slicing algorithm
Dynamic slicing algorithm
Program Source Code
Static Slice
Dynamic Slice

Figure 12. Basic hybrid slicing algorithm

The user can choose to have certain language constructs like loops, function calls handled statically, rather than dynamically. Hence the users can choose between precision, and time and space complexity, depending on their specific computation requirements.

The two extreme scenarios that can occur are: firstly, the algorithm can behave like a traditional dynamic slicing algorithm (with all its advantages and disadvantages), if the user opts to have all language constructs handled dynamically; secondly, if all loops and function calls are handled statically, the algorithm will behave similarly to a traditional static slicing algorithm, leading to a slice that is less precise than a dynamic slice, but with less time and space complexity associated with it.

The first step of the hybrid algorithm, according to Figure 13, is to create the syntax tree of the whole program. Then the execution trace, based on particular inputs is recorded up to the point of interest without recording the user selected language constructs blocks and their statements contained within. Generally speaking, some language constructs that may lead to high run time overhead when the execution trace is recorded can be handled statically, such as loops. These selected blocks will be handled statically by the algorithm. For example, if the user chooses loops to be analyzed statically, the execution trace excluding loops is recorded. If the user selects no static criterion, the hybrid algorithm will compute a pure dynamic slice. On the other hand, if the user selects all available static criteria, the algorithm will compute a pure static slice. After the syntax tree and the execution trace are completed, the hybrid algorithm determines to use static or dynamic slicing algorithm based on the current point of interest and the user-selected options.

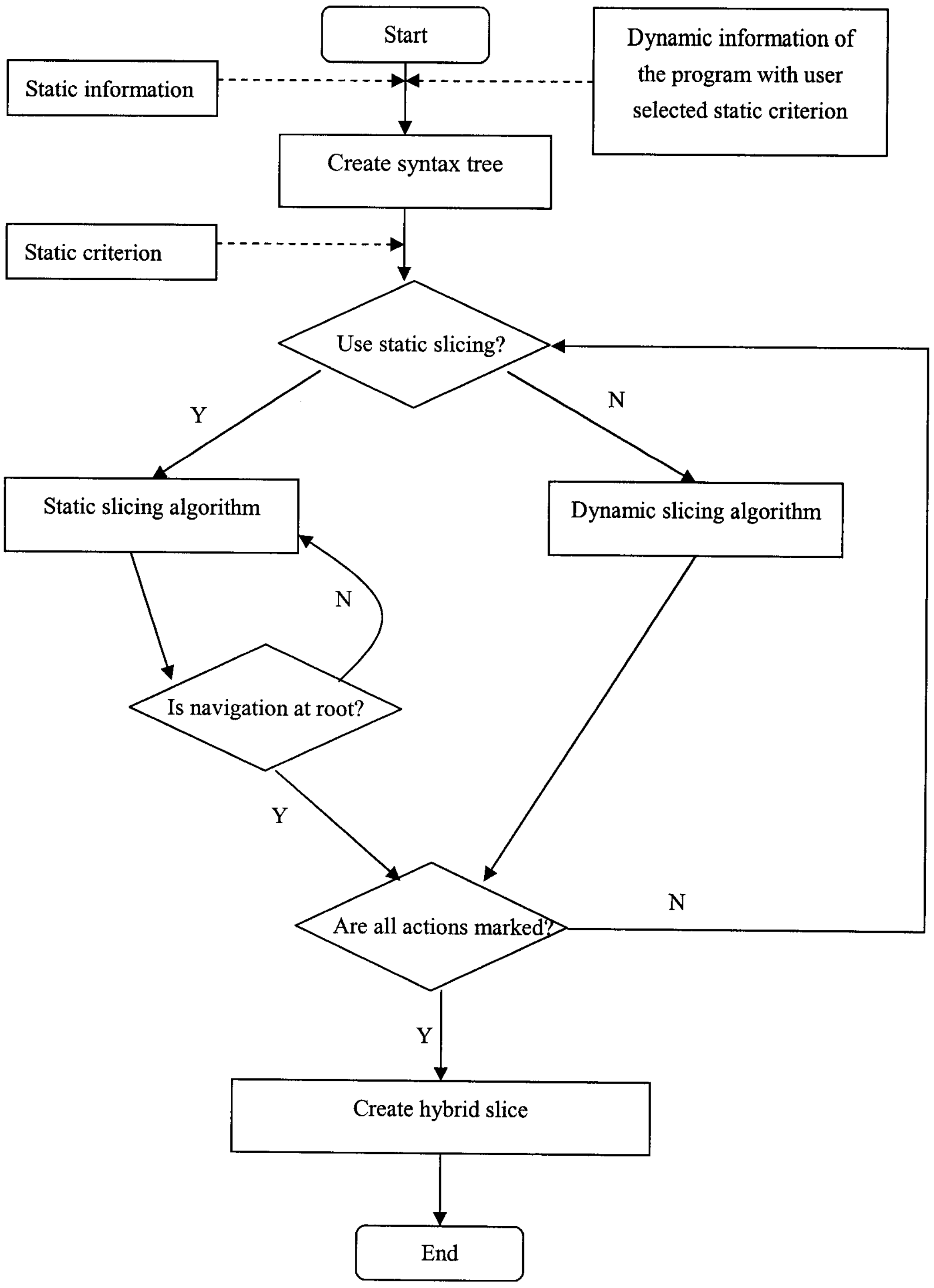
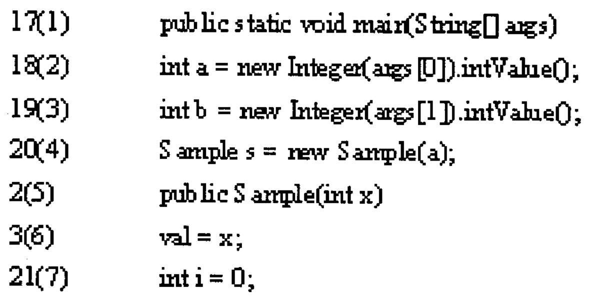


Figure 13. Criterion based hybrid slicing algorithm

If the current point of interest is in one of the user selected language constructs, for example, in a loop, the algorithm will apply static slicing to traverse the syntax tree; in all other cases, the algorithm will use dynamic slicing to traverse the execution trace. Whenever the hybrid algorithm encounters a user selected language construct, dynamic analysis will suspend and the algorithm will switch to the static slicing algorithm. The dynamic slicing algorithm will then resume until the analysis within the selected language construct finishes. For example, if the criterion based hybrid slicing algorithm is used to analyze the program in Figure 1 and the slicing criterion isC = (*a*, 25) and the user selected to analyze loops statically. The syntax tree is the same as that shown in Figure 5 and the execution trace is shown in Figure 14.



```
17(1)     public static void main(String[] args)
18(2)     int a = new Integer(args [0]).intValue();
19(3)     int b = new Integer(args [1]).intValue();
20(4)     Sample s = new Sample(a);
2(5)      public Sample(int x)
3(6)      val = x;
21(7)     int i = 0;
```

Figure 14. Execution trace of hybrid algorithm

After the execution position 7, the hybrid algorithm switches to static analysis without recording the execution trace. The point of interest is now in an user selected construct and the hybrid slicing algorithm starts traversing the syntax tree to find the last definition of variable *a* at node 25. In this example, the root of the syntax tree should be node 22. That means, when the static slicing algorithm encounters node 22 and finishes analyzing it, the hybrid algorithm will suspend the static slicing algorithm and switch to the dynamic slicing algorithm.

# 4 Implementation

Our hybrid slicing algorithm is implemented in the CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) project that was founded with the motivation in mind to address current and future challenges in the comprehension of large and distributed systems, by providing programmers with novel comprehension techniques. These techniques are based on a variety of source code analysis, visualization, and application approaches.

The hybrid slicing algorithm is implemented based on both static and dynamic information. To retrieve static information, the source code has to be parsed first and all parsed information has to be stored after. The CONCEPT project uses JavaC [???] to parse the source code to get the abstract syntax tree of the source code, which is stored in a PostgreSQL database. The execution trace extracted by the tool of ***inst*** is also stored in the database.

Since we do not take the memory address of the variable, it is hard to implement the dynamic slicing algorithm just by tracing the variable along the execution trace as it is done in Korel's algorithm[Kor88]. We had to make modifications to our hybrid slicing algorithm to facilitate these differences accordingly when we use the dynamic information to refine the result of the static slicing algorithm.

As part of this research, we implement the criterion based hybrid slicing algorithm mentioned above. We also modify the instrumentation program to allow for a selective recording of the execution traces based on the user's choice. This leads to a reduced execution trace and therefore to less overhead with respect to space and time complexity involved in the computation of program slices (compared with the traditional dynamic slice). The static slice computed by the static slicing algorithm is then reduced by removing those statements that do not appear in the execution trace. The weakness of our approach compared with the traditional dynamic slicing algorithm is that our approach has to make conservative assumptions with respect to certain dynamic language constructs, e.g. arrays. The reason for this imprecision is caused by the limited information provided by the instrumenter and the recorded information in the execution trace.

## 4.1 Database

We use PostgreSQL, an open source database management system to construct our knowledge base where the static and dynamic information is stored. Our database system [Zha03] includes the tables and the database interface (DBI). First of all, the data about the source program - the Abstract Syntax Tree (AST) is generated by JavaC. Then the whole AST is stored into the database. In order to view and extract the data easily, we create 13 tables in our database system, which respectively correspond to the projects, entities in projects, relationships between entities, java files, imported java files, classes, interfaces implemented by classes, information about classes used in projects, class members (methods, constructors and fields), method arguments, method members, relation codes and type codes. The DBI is an interface between the database and the user. It makes the database transparent to the user who just invokes functions without creating SQL statements to access the database. Using the DBI, we can access the data stored in the 13 tables mentioned above. For example, if we want to get the information about the "min" function of the sample program shown in Figure 1 (assuming the class Sample is in the project SampleProject's SampleFile, and the AST of SampleProject has already been generated and stored in the database), we can find SampleProject in the table about projects first, then find SampleFile in the table about files, thereby we can find the class Sample. After that, we can find function "min" that is a member of the class Sample. Now what we get is the root of the sub-tree representing the whole "min" function. By navigating the sub-tree, we can get all the data about the function.

## 4.2 Selective Execution Trace

The tool that we use to produce the execution trace is ***instr***, a Java package for instrumenting the Java source code. The program requires the following steps to instrument and record the execution trace:

1. Read the source file and parse it into a tree, preserving all comments and white space.
2. Annotate the parse tree with instrumentation, used to monitor the execution of the program.
3. Write the annotated tree to a file that contains Java source code with some additions.

4. Compile the annotated source.

5. Remove the annotations from the source code.

6. Execute the program and collect instrumentation data.

Based on the above steps, ***instr*** can generate the execution trace for the whole program. But what we need is a selective execution trace to avoid some recording overheads. Therefore, we modify ***instr*** to allow for a selective execution trace recording according to the user's choice. In the present implementation, the user's choice is an argument that is used as an input for ***instr***, which does not annotate the statements that satisfy the input. For example, the while loop block will not be annotated when a user chooses to handle while loops statically and as a consequence, the execution trace will not record any executions of while loops. In what follows, we will describe more in detail the underlying static slicing algorithm and some of its implementation issues.

## 4.3 Static Slicing Algorithm

The static slicing algorithm is an important and the most complicated part of the hybrid slicing algorithm implementation, in particular because the algorithm has to support object-oriented programming language constructs. The major implementation tasks of our algorithm are constructing the dependency graph, traversing the graph to get data dependencies and including declarations.

### 4.3.1 Construct the dependency graph in memory

The static slicing algorithm has to construct a dependency graph that has to be traversed to generate the slice. For the algorithm implementation, the graph must be stored in memory at first. In this research, the graph representing control dependencies and function calls is stored in a vector, whose elements are linked lists. The analysis is performed in three phases: conversion of blocks, construction of the basic graph and addition of control dependencies and function calls.

#### 4.3.1.1 Conversion of blocks

Our static slicing algorithm is based on static blocks that package the information that the algorithm needs to analyze the source code, such as used and defined variables and control dependences. In addition, each static block corresponds to a statement in the source code. However, this information is stored in the database separately in different tables making a direct access to some of the information more complicated.

To implement the static slicing algorithm, therefore, conversion of this information from its database representation to the specific algorithm needs was performed. The structure of the static block is shown in Figure 15. So, the first step is to collect and integrate the separate information into complete static blocks.

```
public class StaticBlock implements Constants{
  private int lineNumber;
  private int blockType;
  private Object blockValue;
  private int ctrlDepStatementId; // Direct control dependence
  private Vector usedVariables;
  private Vector defVariables;
  private Vector functionCalls;
  private Vector funDeclarations;
}
```

Figure15. The structure of class StaticBlock

In the program, every static block has an index, referred to a block id that consists of the project, file and statement ids. When a static block is created, it and its index are stored into a hash table – block table where the index is the key and the static block is the value. Later on, we can find the static block using its block id.

#### 4.3.1.2 Constructing the basic graph

As we mentioned before, the graph is stored in the memory as a vector whose elements are linked lists. The basic graph corresponds to the root level representation of the tree, where every linked list is represented only by the head of the sub-tree.

Before the graph is constructed, nodes have to be created. In the program, every node maps one-to-one to a block id in the block table. When a node is created, the corresponding block id is included into the node as a field of node value. After a node is created, it is inserted into the linked list as the head of the list. The basic graph consists of these linked lists that only contain heads. Besides the block id as the value, a node has other elements like status and type.

There exist different types of nodes: regular and function-call nodes. Every regular node corresponds to a real statement of the source code, while a function-call node represents a function call that happens anywhere and might not correspond to a statement. Although the edge of the graph should be used to express a function call, it is hard to define different kinds of edges in a linked list. Furthermore, both regular nodes and function-call nodes are derived from basic nodes. The structure of the class of Node, RegularNode and FunctioncallNode is shown in Figure 16. More details about these two kinds of nodes will be addressed.

```
public class Node {
  private int nodeType;
  private int nodeStatus;
  private BlockId nodeValue;
}

public class RegularNode extends Node implements Constants{
  private boolean isTopNode;
  // Record the number of how many interesting definitions found in current "case" blocks.
  // 'visited but not marked' or 'marked' to be analyzed and included into slice ultimately.
  private int foundInHowManyBranches;
  // Verify if the found interesting defintions are in the same scope when deal with "if...else".
  private int scopeNumber;
}

public class FunctionCallNode extends Node implements Constants{
  private Vector actParameters;
}
```

Figure16. The structures of classes Node, RegularNode and FunctionCallNode

## Add control dependencies and function calls

Control dependencies and function-call relations between nodes will be added into the graph after the basic graph is constructed. Based on the static information from the parser stored in the database, every static block is control dependent on another one (except if they represent a root block). The control dependence is stored in the field of "ctrlDepStatementId" in the class of StaticBlock, from which it is easy to identify the node (called control dependent node) on which other blocks are control dependent. The program traverses every node in the basic graph (that is, the head of every linked list) to find its control dependent node, and then the node is inserted into the linked list beginning with its control dependent node. The nodes in the linked list are then sorted in a descending sequence in terms of "statementId". Note that only the existing node in the basic graph rather than the newly created node (actually there is no a node object newly created) is inserted into the linked list, in order to make the regular node is unique in the graphs.

Adding function-call nodes is different because the node has to remember the actual parameters when the call occurs. The function-call nodes corresponding to the same function are unique in the graph even though they have the same function names and the function might be declared in the same place. Like adding control dependencies, the program extracts the information about the actual parameters and the function's declaration dependency when it accesses a regular node. Based on this information, a new function-call node is created. Then the node is inserted in the end of the linked list whose head is the node where the function is called. If there are more than one functions called in a node, these function-call nodes are sorted in a descending sequence in terms of the locations of the function-call sites. In the graph, every function-call node corresponds to a regular node where the function is defined. Actually, the node value of a function-call node is the block id of the statement where the function is defined.

In what follows we consider a more challenging situation with the actual parameters could be also a function call. In this case, every function-call parameter is extracted and a function-call node is created based on the function-call parameter. Then these function-call nodes are inserted into the linked list representing the call site in an ascending sequence. Let's take a look at the following example.

```
var = fun1(fun2(fun3(a, b), fun4(c, d)), fun5(e, f)).fun6().fun7();
```

In the above sample statement, there is a total of 7 functions-calls and some of them are actual parameters. In the program, all these function-calls are inserted into the linked list whose head is the block id of the statement in the following sequence:

fun7 → fun6 → fun1 → fun5 → fun2 → fun4 → fun3

The source code for the sample program is shown in Appendix 1, which includes three java files: Elevator.java (file1), Panel.java (file2) and Button.java (file3). The number in the first column of the table is the line number. In some cases the line number is same as the statement id when there is only one statement in the line. A part of the graph representing Panel.java is shown in Table 3. The whole table is a vector and each row is an element of the vector. In the graph, the underlined numbers represent function-call nodes, the non-underlined statements correspond to regular nodes. The node "3.9" refers to the file 3's node number 9, which is function "pressed()" defined in Button.java.

### 4.3.2 Traverse the graph to search for the last definition

A depth-first-search algorithm is used in traversing the graph in implementation of the static slicing algorithm. Usually the traversal process starts from the point of interest defined by the user to search for the last definition of the variable of interest also defined by the user. At first, the user enters the project name, file name and line where the point of interest is located, as well as the variable name of interest. Then the program identifies the node according to the input and validates if the variable of interest is used in there. If not, the program will exit with an error message being returned, because the program assumes that the starting point should be the place where the variable of interest is used.

After determining the starting node, the program finds the point where to start the traversal. Note that there could be more than one place where the starting node is found. For example, if the starting node like node 3.9 in Table 3 is the entry point of a function and this function is called in different places. That is, the function has multiple related function-call nodes in different linked lists. Conservatively, the static slicing algorithm will take all possibilities into account. Thus, all of these places will be found and remembered by a vector. Then the program will traverse beginning from them one by one except for some special cases that will be addressed in the following sections.

Table 3. Dependency graph of program Panel.java

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | → | 1 | | | | | | | | | | | | | | | | | | |
| 1 | → | 34 | → | 29 | → | 24 | → | 22 | → | 17 | → | 11 | → | 9 | → | 4 | → | 3 | → | 2 |
| 2 | | | | | | | | | | | | | | | | | | | | |
| 3 | | | | | | | | | | | | | | | | | | | | |
| 4 | → | 7 | → | 6 | → | 5 | | | | | | | | | | | | | | |
| 5 | | | | | | | | | | | | | | | | | | | | |
| 6 | → | 3.5 | | | | | | | | | | | | | | | | | | |
| 7 | → | 8 | | | | | | | | | | | | | | | | | | |
| 8 | → | 3.5 | | | | | | | | | | | | | | | | | | |
| 9 | → | 10 | | | | | | | | | | | | | | | | | | |
| 10 | → | 3.14 | | | | | | | | | | | | | | | | | | |
| 11 | → | 16 | → | 13 | → | 12 | | | | | | | | | | | | | | |
| 12 | | | | | | | | | | | | | | | | | | | | |
| 13 | → | 14 | | | | | | | | | | | | | | | | | | |
| 14 | → | 15 | → | 3.9 | | | | | | | | | | | | | | | | |
| 15 | | | | | | | | | | | | | | | | | | | | |
| 16 | | | | | | | | | | | | | | | | | | | | |
| 17 | → | 18 | | | | | | | | | | | | | | | | | | |
| 18 | → | 20 | → | 19 | → | 3.9 | | | | | | | | | | | | | | |
| 19 | | | | | | | | | | | | | | | | | | | | |
| 20 | → | 21 | | | | | | | | | | | | | | | | | | |
| 21 | | | | | | | | | | | | | | | | | | | | |
| 22 | → | 23 | | | | | | | | | | | | | | | | | | |
| 23 | → | 3.7 | | | | | | | | | | | | | | | | | | |
| 24 | → | 28 | → | 25 | | | | | | | | | | | | | | | | |
| 25 | → | 26 | | | | | | | | | | | | | | | | | | |
| 26 | → | 27 | → | 3.9 | | | | | | | | | | | | | | | | |
| 27 | | | | | | | | | | | | | | | | | | | | |
| 28 | | | | | | | | | | | | | | | | | | | | |
| 29 | → | 33 | → | 30 | | | | | | | | | | | | | | | | |
| 30 | → | 31 | | | | | | | | | | | | | | | | | | |
| 31 | → | 32 | → | 3.9 | | | | | | | | | | | | | | | | |
| 32 | | | | | | | | | | | | | | | | | | | | |
| 33 | | | | | | | | | | | | | | | | | | | | |
| 34 | → | 38 | → | 35 | | | | | | | | | | | | | | | | |
| 35 | → | 36 | | | | | | | | | | | | | | | | | | |
| 36 | → | 37 | → | 3.9 | | | | | | | | | | | | | | | | |
| 37 | | | | | | | | | | | | | | | | | | | | |
| 38 | | | | | | | | | | | | | | | | | | | | |

Since the static slicing algorithm is backward, the traversal should go backward from the starting point. However, this might not always be the case. In the case that the starting point is a loop statement, the traverse has to go forward to consider the statements after the starting point. These special cases will be discussed in detail in the following part.

Assuming the starting point is identified, the algorithm will go to the node right next to the starting point. The depth-first-search algorithm marks all relevant nodes be visited as if follow the control flow to backward traverse the source code statement by statement until the last definition of the variable of interest is found. In the program, the traverse process is implemented by three overloaded functions: "searchLastDef(Node, Variable)", "searchLastDef(Node, Variable, LinkedList)" and "searchLastDef(Variable, Node)". The first step is to recursively find the linked list that includes starting point. In a second step the search function handles the search within the list found in the first step. Note that the starting point is changeable during the traversal, from the initial position determined by the user's input to the head of the linked list including the previous starting point. The process continues recursively until the last definition is found or the top of the search range is encountered. The concept of the search range is to be a boundary of the search that will be addressed in detail in the following section. The second search function is in charge of handling the search within the current list by calling the third search function. The third search function recursively deals with the traversal beginning from the current node, and different special cases including "if ... else", "switch ... case" and parameters in the function call.

There could be some branches in the traversal when the inside-out inter-procedural call is encountered. In this situation, all possible paths have to be considered at first. However, after the analysis, some of these paths can be eliminated if they do not contain any relevant statements. The algorithm uses for this, a stack "tempMarkedNodeVectors" to record all of possible paths temporarily. Every time a branch is encountered, a new vector "tempMarkedNodes" is created and pushed onto the stack and every node traversed afterward will be recorded into the vector. If there is nothing found after the last node is visited, the latest vector in the stack will be popped and deleted. On the contrary, all elements of the stack will be transferred to the collection "markedNodes" as relevant nodes to be analyzed later.

### 4.3.3 Include declaration information

The last step of computing the slice is to take the declarations into account, such as information about the variables, classes and functions involved in the relevant nodes in "markedNodes". Doing so can make the slice executable.

**Variable declare information**

In sample program "panel.java", if the statement 8 is relevant and included in "markedNodes", the declaration of the variable "pbutton" – the statement 3 – has also to be included.

**Class declare information**

Based on the previous example, if statement 3 is included, the declaration of class "Button" has to be included. That is, the statement 1 in "button.java" should be included. Obviously, it is not necessary to include the whole definition of the class.

**Function declaration information**

The function declaration information has to be extracted and analyzed. Take the same Example that the function (constructor) "Button()" is called in statement 8, and statement 8 is in the slice. So the declaration of this function should be included into the slice.

### 4.3.4 Some special cases

As we mentioned in section 3, there are some special cases in traversing the graph do not strictly follow the control flow. These special cases include "if … else", "switch … case", loop and polymorphism. In wat follows, we illustrate how these special cases are handled in the implementation.

#### 4.3.4.1 Conditional branch

Unlike dynamic slicing algorithm, static slicing algorithm has to consider all possibile paths when it analyzes the conditional branches, because the algorithm does not know which path the program takes.

**"IF… ELSE"**

"If … else" is a conditional statement that is often used in the program. There are four kinds of cases that the static slicing algorithm has to handle: traverse from the true case, traverse from the false case, the last definition found in the true case and the last definition found in the false case.

- Traverse from the true case

In this case, the static slicing algorithm just traverses backward the graph as a normal case. The traverse from node 15 in Table 3 is such an example.

- Traverse from the false case

When traverse starts from the false case to search for the last definition of the variable of interest, the true case of the same conditional statement will be ignored. In the implementation, the program gets the "if" node on which the current node is control dependent and traverse from the "if" node immediately after the traversal in the false case has finished. For instance, the traverse starting from node 21 will skip node 19.

- The last definition is found in a true case

If the last definition of the variable of interest is found in the true case, the algorithm must go to the false case to do another search because the static slicing algorithm should not know which case is exactly executed. In the implementation, firstly, the program can get the "if" node on which the current node is control dependent when the last definition of the variable of interest is found in the current node. Then the scope number of the current node, which can be extracted from the database is recorded in the field "scopeNumber" of the "if" node. And the search will continue in the linked list where the "if" node is the head until another last definition of the variable of interest is found. Then the program compares the scope number of the current node with another one recorded in the "if" node. If they are different, the search will stop. Otherwise, the search will continue. In sample program "Elevator.java", for example, if the last definition of variable "current_floor" is found in static block 31, the search will continue to static block 33 rather than stop. In case there is nothing found in static block 33, the algorithm will go on to search backward from static block 30.

- The last definition found in the false case

Similar to the case that the last definition is found in the true case, if the last definition is found in the false case, the algorithm has to do another search in the true case. The search must continue if nothing is found in the true case. **"SWICH … CASE"**

"Switch … case" is handled in the implementation similar to "if … else". There are still two types of cases to be considered: traverse from a case block and the last definition found in a case block.

- Traverse from a case block

Normally, if the traversal starts from a case block, the algorithm will ignore other case blocks and go to the "switch" statement directly. In the implementation, the program gets the "switch" node on which the current node in a case block is control dependent and continues the traverse from the "switch" node immediately after the traversal in the case block has finished.

- The last definition found in a case block

If the last definition of the variable of interest is found in a case block, the algorithm must continue to search in every other case block. The traversal must continue outside the "switch" block if nothing is found in any one of the case blocks. In the implementation, this is supported by field "foundInHowManyBranches" of the "switch" node, which is used to record how many case blocks where the last definition is found. When the "switch" node is eventually encountered, comparison between the value of this field and the number of case blocks that the "switch" node includes will check if every case block has the last definition of the variable of interest. If not, the traverse will continue.

#### 4.3.4.2 Loops

There are three kinds of loops in Java: for loop, while loop and do loop. In the static slicing algorithm, the ways to deal with these loops are similar. No matter what kind of loop, there are two following situations that have to be considered in the algorithm.

**Traverse from the loop node**

The loop statement is a "for" statement, "while" statement or "do" statement that represents the beginning of a loop. The loop node corresponds to the loop statement. In the implementation, if the current node is a loop node, the program will call the third search function "searchLastDef(Variable, Node)" to search the last definition of the variable of interest in the linked list with the loop node as the head before it calls the second search function "searchLastDef(Node, Variable, LinkedList)". For example, in Table 3, the traverse starting from node 13 has to visit node 14, 15 and 3.9 before going to node 12.

**Traverse from the statement within the loop block**

Since the static slicing algorithm cannot know how many times the loop iterates, conservatively, the algorithm assumes that it does at least one time. The program traverses the graph until the loop node is encountered. Then the program continues the search to the traversal of the loop node. In Table 3, for example, the traverse starting from node 14 must go to node 15 after node 13 is encountered.

#### 4.3.4.3 4.3.4.3 Inter-procedural traverse

As we mentioned in section 3, there are two kinds of inter-procedural traversals: outside-in and inside-out.

**Outside-in inter-procedural traversal**

- Record the actual parameters

As we discussed in the previous sections, the function-call nodes in the graph are used to record the actual parameters when function calls happen. Originally, the information about the actual parameters is from the database. The information is stored into the relative function-call node when the graph is constructed. For example, in the sample program Elevator.java, the actual parameters of the function call – "top_floor" and "current_floor" - happens in the static block 22 are recorded into the function-call node 2.34 that is in the linked list 22. Likewise, the actual parameters of the same function call – "current_floor" and "1" - happens in the static block 28 are recorded into the function-call node 2.34 in the linked list 28. These two nodes 2.34 are different objects with same declare-dependency keys, that is, correspond to the same regular node 2.34

It is straightforward to just record the actual parameter directly if it is a variable. However, the case becomes complicated if the actual parameter is another function call or even a nested function call whose parameter is also a function call. In this case, the program must sort these function calls in a certain order that we mentioned in section 4.3.1.3. The traversal will go inside every function one by one.

- Pass the actual parameters to the formal parameters

In order to pass correct actual parameters to their corresponding formal parameters, actual parameters have to be stored in a sequence in which formal parameters are declared. When the traverse encounters a function-call node, the algorithm will go to the corresponding regular node where the function is declared according to the declare dependency keys.

In the process of searching for the last definition of the variable of interest, only those statements relevant to the variable of interest should be considered. The inter-procedural traverse is not an exception. Therefore, only in such a case that the actual parameter is same as the variable of interest that is a reference of an object, it is necessary to analyze what happens to the parameter within the definition of the function. Hence, before the analysis, there should be a verification to check if 1) the actual parameters include the variable of interest and 2) the variable of interest is a reference of an object. The reason is that in Java, the parameter's value could be permanently changed by a function only when the parameter is the reference of an object.

In the implementation of the algorithm, the program checks the actual parameters recorded in the function-call node to see if one of them is same as the variable of interest. In the algorithm, two variables are same if they have same names and declare dependency keys. If yes, and the variable of interest is a reference of an object, the variable of interest will be temporarily replaced by the corresponding formal parameter when the traverse is within the called function. Otherwise, if the actual parameters do not include the variable of interest or the variable of interest is not a reference of an object, there is nothing to do before traversing the called function.

**Inside-out inter-procedural traverse**

- When the variable of interest is of a global variable

  In this case, the traversal is handled normally without exchanging between parameters.

- When the variable of interest is a local variable

  The traversal will be confined within the function.

- When the variable of interest is a formal parameter

  The way to handle the case that the variable of interest is a formal parameter is contrary to the way mentioned in the outside-in inter-procedural traversal. When the node where the function is called is found, the formal parameter is replaced by its corresponding actual parameter regardless of that the formal parameter is a reference of an object or not, and the traversal continues outside the function.

#### 4.3.4.4 Polymorphism

It is very difficult for a static slicing algorithm to gain some dynamic information such as dynamic-binding. Therefore, polymorphism has to be handled by the static slicing algorithm very conservatively, that is, all possibilities must be considered unless we can extract the information about the object's type to know which path will be taken.

# 5 Conclusions and future work

In this thesis, we presented our hybrid slicing algorithm that is based on the notion of removable blocks and can compute executable slices for Java programs.

## 5.1 Summary of contributions

The contributions of this research can be summarized as the following:

### 5.1.1 A survey of slicing concepts

Various slicing algorithms have been proposed since Mark Weiser introduced the concept of program slicing in 1979. This thesis presents a literature review of program slicing based on the classification – static, dynamic and hybrid program slicing. For each of these slicing categories we present several approaches and discuss their advantages and disadvantage.

### 5.1.2 A modified version of the instrumentation program

To make the static slice more precise, similar to some other hybrid slicing algorithm, our hybrid slicing algorithm uses the execution trace to refine the result of the static slicing algorithm. In the implementation, a third party's tool called ***instr*** is involved to create the execution trace. On the other hand, our hybrid slicing algorithm must overcome the drawback of the dynamic slicing algorithm that it sometimes incurs high runtime overheads. Besides, as a criterion based hybrid slicing algorithm, our hybrid slicing algorithm has to let the user determine what language construct is handled statically or dynamically. Therefore, we modified ***instr*** to let it record the execution trace selectively based on the user's inputs. When the user thinks the loops in a program to be analyzed could generate a high runtime overhead, he can input an code corresponding to the loop as a criterion of the hybrid slicing algorithm. The modified version of ***instr*** will not record the execution trace that happens within the loop.

### 5.1.3 A static object-oriented backward slicing algorithm for Java programs

Java programs are the target that our hybrid slicing algorithm analyzes. As an object-oriented programming language, Java has some common features of object-oriented programming languages, such as inheritance, polymorphism and so on. Our hybrid slicing algorithm considers them when the static slicing algorithm is implemented.

We use JavaC to extract the static information of a Java program, which includes classes' definitions and usages, inheritance between classes, function definitions and function calls, and so on. Based on this information, the static slicing algorithm can take most object-oriented features of the Java program into account.

### 5.1.4 The introduction of a criterion based hybrid slicing algorithm

This thesis introduces a criterion based hybrid slicing algorithm for object-oriented programs, which can switch between the static and the dynamic slicing algorithms based on the user's inputs. It makes the static slice more precise and overcomes to some extent, the disadvantage of dynamic slicing that it may be too expensive when it handles some language constructs.

### 5.1.5 The implementation of the hybrid slicing algorithm

Our hybrid slicing algorithm is implemented in CONCEPT project in Java. It uses the static information and the selective execution trace that stored in a PostgreSQL database to compute the hybrid slice. According to the architecture of CONCEPT, the hybrid slice will be used by applications like visualization and metrics.

## 5.2 Limitations

Our current hybrid slicing algorithm is based on some assumptions and limitations that will be addressed as a part of future work. Some of these limitations include the handling of polymorphisms and arrays.

### 5.2.1 Handling of polymorphisms

Even though JavaC can extract some information about the polymorphism when it is obvious, some situations like that objects are expressed by elements of arrays cannot be dealt with. That means our hybrid slicing algorithm can partially resolve the issue of the polymorphism. For those cases that cannot be handled, our hybrid slicing algorithm has to conservatively consider all possibilities.

### 5.2.2 Handling of arrays

When the index of a array's elements is a variable and the variable of interest is a certain element of the array, since the static slicing algorithm does not the value of the index, the static slicing algorithm has to analyze all elements of the array leading to an imprecise slice.

### 5.2.3 Line based execution traces

The execution trace created by the tool ***instr*** is only a set of line numbers. That is, the execution trace is line based not statement based. For the case that there are more than one statement in a line, instr may get an incorrect execution trace that causes a imprecise slice.

## 5.3 Future work

The current hybrid slicing algorithm will be refined to overcome the limitations addressed above. Also the algorithm has to be integrated with other parts of the CONCEPT environment like visualization, testing and debugging. Future work should also extend the application of the hybrid algorithm for distributed and concurrent systems.

## References

[Agr90] Agrawl, H., and Horgan, J. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation* (1990), pp. 246-256. *SIGPLAN Notices* 25(6).

[Agr92] Agrawal, H. Towards automatic debugging of Computer Programs. PhD thesis, Purdue University, 1992.

[Aho96] Aho, A., Sethi, R., Ullman, J. Compilers: Principles, Techniques and Tools. *Addison-Wesley Publishing Company,* Reading, Massachusetts, 1996.

[Bat93] Bates, S., and Horwitz, S. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* (Charleston, SC, 1993), pp. 384-396.

[Ber85] Bergeretti, J.-F., and Carre, B. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems 7,* 1 (1985), 37-61.

[Boy79] Boyer, R. S., and Moore, J. S. A Computational Logic. Academic Press, New York, 1979.

[Bra95] Brandis, M. Optimizing Compilers for Structured Programming Languages. Dissertation, ETH Zurich, 1995.

[Can98] Canfora, G., Cimitile, A., and De Lucia, A. Conditioned program slicing. *Information and Software Technology*, vol. 40, no. 11/12, 1998, pp.595-607.

[Cho91] 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 (1991), 491-530.

[Cow88] Coward, P. D. Symbolic execution systems – a review. *Software Engineering Journal,* vol. 3, no. 6, 1988, pp. 229-239.

[Dan00] Danicic, S., Fox, C., Harman, M., and Hierons, R. ConSIT: a conditioned program slicer. In *Proceedings of International Conference on Software Maintenance,* S.Jose, USA, 2000, IEEE CS Press, pp. 216-226.

[Due92] Duesterwald, E. Gupta, R., and Soffa, M. L. Rigorous data flow testing through output influences. In *Proceedings of the 2nd Irvine Software Symposium.* (1992) 131-145.

[Fer87] Ferrante, J., Ottenstein, K., and Warren, J. The program dependence graph and its use in optimization. *ACM transactions on Programming Languages and Systems 9,* 3(1987), 319-349.

[Gup92] Gupta, R., Harrold, M., and Soffa, M. An approach to regression testing using slicing. In *proceedings of the Conference on Software Maintenance* (1992), pp. 299-308.

[Gup97] Gupta, R., Soffa, M., and Howard, J. Hybrid Slicing: Integrating Dynamic Information with Static Analysis. *ACM Transactions on Software Engineering and Methodology,* 6(4), October 1997, pp. 370-397.

[Hor88] Horwitz, S., Prins, J., and Reps, T. Integrating non-interfering versions of programs. In *Conference Record of the ACM SIGSOFT/SIGPLAN Symposium on Principles of Programming Languages* (1988), pp. 133-145.

[Hor89] Horwitz, S, Pfeiffer, P., and Reps, T. Dependence analysis for pointer variables. In *Proceedings of the ACM 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, 1989). *SIGPLAN Notices* 24(7).

[Hor90a] Horwitz, S. Identifying the semantic and textual difference between two versions of a program. In *proceedings of the ACM SIGPLAN'90 Conference on Program Language Design and Implementation* (White Plains, New York, 1990), pp. 234-245. *SIGPLAN Notices* 25(6).

[Hor90b] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12,* 1(1990), 26-61.

[Kam93] Kamkar, M., Fritzson, P., and Shahmehri, N. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *proceedings of the Conference on Software Maintenance* (Montreal, Canada, 1993), pp. 386-395.

[Kin76] King, J. C. Symbolic execution and program testing. *Communications of the ACM,* vol. 19, no. 7, 1976, pp. 385-394.

[Kor88] Korel, B., and Laski, J. Dynamic program slicing. *Information Proceeding Letters 29,* 3 (1988), 155-163.

[Kor92] Korel, B., and Ferguson, R. Dynamic slicing of distributed programs. *Applied Mathematics & Computer Science Journal,* 2(2), 1992, pp. 199-215.

[Kor98] Korel, B., and Rilling, J. CASE and dynamic program slicing in software maintenance. *International Journal of Computer Science and Information Management,* June 1998.

[Lar96] Larsen, L., and Harrold, M. Slicing Object-oriented Software. In *Proceedings of the 18th International Conference on Software Engineering* (Berlin, 1996), pp. 495-505.

[Liv94] Livadas, P. E., Croll, S. A New Algorithm for the Calculation of Transitive Dependences. Technical Report, Computer and Information Science Department, University of Florida, 1994.

[Liv95] Livadas, P. E., Johnson, T. An Optimal Algorithm for the Construction of the System Dependence Graph. Technical Report, Computer and Information Science Department, University Florida, 1995.

[Mil88] Miller, B., and Choi, J.-D. A mechanism for efficient debugging of parallel programs. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, 1988), pp. 135-144. *SIGPLAN Notices* 23(7).

[Net94] Netzer, R., and Weaver, M. Optimal tracing and incremental reexecution for debugging long-running programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* (1994). ACM, New York, 313-325.

[Nis99] Nishimatsu, A., Jihira, M., Kusumoto, S., and Inoue, K. Call-Mark Slicing: An Efficient and Economical Way of Reduce Slice. ICSE'99 Los Angeles CA.

[Ott84] 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* (1984), pp. 177-184. *SIGPLAN Notices* 19(5).

[Rep89] Reps, T., and Bricker, T. Illustrating interference in interfering versions of programs. In *Proceedings of the Second International Workshop on Software Configuration Management* (Princeton 1989), pp. 46-55. *ACM SIGSOFT Software Engineering Notes* Vol.17 No.7.

[Rep94] Reps, T., Sagiv, M., and Horwitz, S. Interprocedural dataflow analysis via graph reachability. Report DIKU TR 94-14, University of Copenhagen, Cepenhagen, 1994.

[Ril01] Rilling J., and Karanth, B. A Hybrid Program Slicing Framework. *IEEE International Workshop on Source Code Analysis and Manipulation SCAM 2001,* Florence, Italy, November 2001.

[Sch95] 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,* Portland, USA, December 1995, pp. 41-48.

[Tip95] Tip, F. A survey of program slicing techniques. *Journal of Programming Language,* vol. 3, 1995, pp. 121-189.

[Wei82] Weiser, M. Programmers use slices when debugging. *Communications of the ACM 25*, 7 (1982), 446-452.

[Wei84] Weiser, M. Program slicing. *IEEE Transactions on software Engineering 10*, 4 (1984), 352-357.

[Zha03] Zhang, Y.G. Automatic design pattern recovery. Master thesis, Concordia University, 2003.

# Appendix 1. Sample program for the static slicing algorithm

```
0     package elevator;
      import java.io.*;
1     public class Elevator {
2         private boolean OPEN = true, CLOSED = false; //Door
3         private int current_floor, next_floor;
4         final int top_floor;
5         Panel panel;
6         private final boolean UP = true, DOWN = false; //Direction
7         private boolean door, direction;

8         public Elevator(int num) {
9             top_floor = num;
10            panel = new Panel(num);
11            door = OPEN;
12            direction = UP;
13            current_floor = 1;
          }
14        public void closedoor() {
15            door = CLOSED;
16            System.out.println("Closing the door!");
          }
17        public void move() {
18            if (direction == UP) {
19                if (current_floor == top_floor)
20                    direction = DOWN;
21                else
22                    if (!panel.demandedInThisDirection(top_floor, current_floor))
23                        direction = DOWN;
              }

24            if (direction == DOWN) {
25                if (current_floor == 1)
26                    direction = UP;
27                else
28                    if (!panel.demandedInThisDirection(current_floor, 1))
29                        direction = UP;
              }

30            if (direction == UP) {
31                current_floor = panel.findMiniumFloor(top_floor, current_floor);
32            } else
33                current_floor = panel.findMaxiumFloor(current_floor, 1);

34            System.out.println("Now moving to floor: " + current_floor);
35            System.out.println("Arrived at floor: " + current_floor);

36            panel.buttonOff(current_floor);
37            openDoor();
          }

38        public void openDoor() {
39            door = OPEN;
40            System.out.println("Door is open!");
41            System.out.println();
          }

42        public void prompt() throws IOException {
43            int num = 0;
44            String strnum;

45            BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
46            System.out.println(
47                "Enter the floor you want to go: (between 1 and " + top_floor + ")");
48            while ((strnum = in.readLine()) != null) {
49                num = Integer.parseInt(strnum);

50                if (num == current_floor)
51                    System.out.println("We are at the current floor!");
52                else
53                    if (num == 13) //there is no 13 floor
54                        System.out.println(
                              "Sorry, this floor is not existed. Please press a button again!");
55                    else
56                        if (num > 0 && num <= top_floor) {
57                            panel.pressButton(num);
58                        } else
59                            if (num == 0) { //go
60                                if (panel.howmany() == 0)
61                                    System.out.println(
                                          "Nobody demande, evelator is stopping, please press a button before
closing the door!");
62                                else {
63                                    closedoor();
64                                    move();
                                  }
65                            } else
66                                if (num == -1)
67                                    break;
68                                else
69                                    System.out.println("Invalid input, repeat again!");
70                System.out.println(
                      "Enter the floor you want to go: (between 1 and " + top_floor + ")");
              } //end of while

71            System.out.println("Good-bye!");
          }

72        public static void main(String[] args) throws IOException {
73            final int number = 15; //constant
74            Elevator Otis = new Elevator(number);
```

```
            System.out.println("This is a simulator of an elevator,");
            System.out.println("0: close the door and go -1: terminate the program");
            System.out.println("Start the program now!");
            System.out.println();
            Otis.prompt();
        }
}
package elevator;
import java.util.*;

public class Panel {

        private int number_of_buttons;
        private Button[] pbutton;

        public Panel(int num) {
            number_of_buttons = num;
            pbutton = new Button[number_of_buttons];
            for (int i = 0; i < pbutton.length; i++) {
                pbutton[i] = new Button();
            }
        }

        public void buttonOff(int num) {
            pbutton[num - 1].turnoff();
        }

        public int howmany() {
            int count = 0;
            for (int i = 0; i <= number_of_buttons - 1; i++) {
                if (pbutton[i].pressed())
                    count++;

            }
            return count;
        }

        public boolean on(int num) {
            if (pbutton[num - 1].pressed())
                return true;
            else
                return false;
        }

        public void pressButton(int num) {
            pbutton[num - 1].press();
            // System.out.println("the NO."+num+" is pressed!");
        }

        public int findMiniumFloor(int upFloor, int downFloor) {
            for (int i = downFloor; i <= upFloor; i++) {
                if (pbutton[i - 1].pressed())
                    return i;
            }
            return 0;
        }

        public int findMaxiumFloor(int upFloor, int downFloor) {
            for (int i = upFloor; i >= downFloor; i--) {
                if (pbutton[i - 1].pressed())
                    return i;
            }
            return 0;
        }

        public boolean demandedInThisDirection(int upFloor, int downFloor) {
            for (int i = downFloor; i <= upFloor; i++) {
                if (pbutton[i - 1].pressed())
                    return true;
            }
            return false;
        }
}

package elevator;

/*Class Button*/

public class Button{

        private final boolean NOT_LIT=false;
        private final boolean LIT=true;
        private boolean state;

        //methods
        public Button() {
            state = NOT_LIT;
        }

        public void press() {
            state = LIT;
        }

        public boolean pressed() {
            if (state == LIT)
                return true;
            else //if(state.toString() =="not_lit")
                return false;

        }
        public void turnoff() {
            state = NOT_LIT;
        }
}
```