A Novel Approach of Forward Dynamic Slicing

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
In
Software Engineering

Submitted By
Deepak Goyal
(800931006)

Under the supervision of:

Dr. Rajesh Bhatia
Professor, CSED
DCRUST, Murthal (Sonepat)

Mr. Karun Verma
Assistant Professor, CSED
Thapar University

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2011

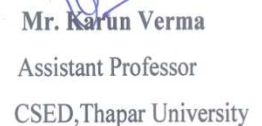
CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"A Novel Approach of Forward Dynamic Slicing "**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Rajesh Bhatia and Mr. Karun Verma** and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Signature
Deepak Goyal

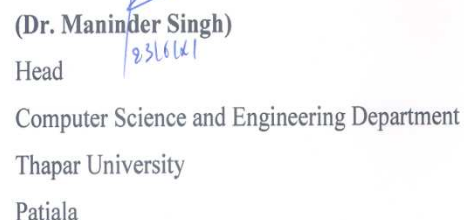
This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

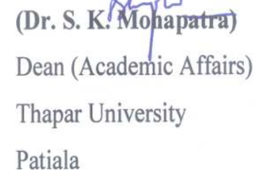


Dr. Rajesh Bhatia
Professor
CSED, DCRUST Murthal (Sonepat)

Mr. Karun Verma
Assistant Professor
CSED,Thapar University

Countersigned by





(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

i

Acknowledgement

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life.

This work would not have been possible without the encouragement and able guidance of my supervisor Dr. Rajesh Bhatia and Mr. Karun Verma. I thank my supervisors for their time, patience, discussions and valuable comments. Their enthusiasm and optimism made this experience both rewarding and enjoyable.

I am equally grateful to Dr. Maninder Singh, Associate Professor & Head, Computer Science & Engineering Department, a nice person, an excellent teacher and a well – credited researcher, who always encouraged me to keep going with work and always advised me with his invaluable suggestions.

I will be failing in my duty if I don't express my gratitude to **'**Dr. Abhijit Mukherjee', Director of the University**,** for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

I am also thankful to the entire faculty and staff members of Computer Science and Engineering Department for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable.

Last but not least, I would like to thank my family whom I dearly miss and without whose blessings none of this would have been possible. To my parents, I own thanks for their wonderful love and encouragement. I would also like to thank my brother, since he insisted that I should do so. I would also like to thank my close friends for their constant support.

Deepak Goyal

(800931006)

ABSTRACT

Program slicing is a deduction technique, which identifies relevant statements that are related to given function or variable at the point of interest in given program. It is used to decompose and filter a large program to restrict the focus on specific parts of program. Program slicing is used in various software engineering activities including program understanding, software testing, software debugging, software maintenance, complexity measurement and reverse engineering etc.

Dynamic slicing is a type of program slicing, which contains all statements that actually affect the value of a variable at the point of interest for a particular execution of the program and forward slice contains all statements that might be influenced by the variable.

A new approach of forward dynamic slicing is introduced in this thesis, which computes forward dynamic slices of programs at execution time without recording program execution trace. So this approach reduces space complexity problem, which occurs due to the storage of program‘s execution trace. The approach presented in this thesis has been implemented in C# on window platform.

Table of Contents

CERTIFICATE	i
ACKNOWLEDGEMENT	ii
ABSTRACT	iii
TABLE OF CONTENTS	iv
LIST OF FIGURES	vi
LIST OF ABBREVIATIONS	viii
Chapter 1 Introduction	**1**
1.1 Introduction	1
1.2 Background	1
1.3 Fundamental Concept	2
1.3.1 Program Slicing	2
1.3.2 Slicing Criteria	2
1.3.3 Type of Slices	2
1.3.3.1 Static Slicing	3
1.3.3.2 Dynamic Slicing	4
1.3.4 Direction of Slicing	6
1.3.4.1 Backward Slicing	6
1.3.4.2 Forward Slicing	7
1.3.5 Level of Slicing	8
1.3.5.1 Intraprocedural Slicing	8
1.3.5.2 Interprocedural Slicing	8
1.3.6 Execution Trace	9
1.3.7 Variable Slice	9
1.3.8 Expression Slice	9
1.3.9 Statement Slice	9
1.3.10 Slicing Operation	9
1.3.11 Dependence	11
1.3.11.1 Data Dependence	11
1.3.11.2 Control Dependence	11
1.4 Tools of Program Slicing	14
1.5 Difference Between Static And Dynamic Slice	16

1.6 Application of Slicing 17
1.7 Structure of Thesis 18
Chapter 2 Literature Survey **19**
2.1 Static Slicing Algorithms 19
2.1.1 PDG Based Algorithms 19
2.1.2 SDG Based Algorithms 21
2.2 Dynamic Slicing Algorithms 25
2.2.1 Dynamic Backward Slicing 25
2.2.2 Dynamic Forward Slicing 27
Chapter 3 Problem Statement **32**
Chapter 4 Proposed Algorithm of Forward Dynamic Slicing **34**
4.1 Proposed Algorithm 34
4.2 Experimental Setup 36
Chapter 5 Implementation and Experimental Result **37**
Chapter 6 Conclusion and Future Work **42**
6.1 Conclusion 42
6.2 Future Work 42

References **43**
List of Papers Presented **46**

List of Figures

Figure-1.1 Sample Program 3
Figure-1.2 Static Slice of Sample Program in Figure-1.1 4
Figure-1.3 Sample Program 5
Figure-1.4 Dynamic Slice of Sample Program in Figure-1.3 6
Figure-1.5 Sample Program 7
Figure-1.6 Backward Slice of Sample Program in Figure-1.5 7
Figure-1.7 Forward Slice of Sample Program in Figure-1.5 8
Figure-1.8 Slice Operations 10
Figure-1.9 Sample Program of Fibonacci Series 12
Figure-1.10 Dependence Graph of Sample Program in Figure-1.9 13
Figure-1.11 Sample Program 14
Figure-1.12 Backward Slice of Sample Program in Figure-1.11 15
Figure-1.13 Sample Program 15
Figure-1.14 Forward Slice of Sample Program in Figure-1.13 16
Figure-1.15 Sample Program for introducing difference between static and dynamic slice 17
Figure-2.1 Sample Program for Introducing PDG 20
Figure-2.2 Symbols used in PDG 20
Figure-2.3 PDG of Sample Program in Figure-2.1 20
Figure-2.4 Static Slice Using PDG 21
Figure-2.5 Sample Program for introducing SDG 23
Figure-2.6 Symbols used in SDG 23
Figure-2.7 SDG of Sample Program in Figure-2.5 24
Figure-2.8 Sample Program for DDG 26
Figure-2.9 DDG of Sample Program in Figure-2.8 27
Figure-2.10 Structure program marked with removable blocks 28
Figure-5.1 Sample Program 37
Figure-5.2 Dynamic Forward Slice of Sample Program in Figure-5.1 38
Figure-5.3 Dynamic Forward Slice of Sample Program in Figure-5.1 38
Figure-5.4 Dynamic Forward Slice of Sample Program in Figure-5.1 39
Figure-5.5 Sample Program 39

Figure-5.6 Dynamic Forward Slice of Sample Program in Figure-5.5 40
Figure-5.7 Dynamic Forward Slice of Sample Program in Figure-5.5 40

List of Abbreviations

CD : Control Dependence

DD : Data Dependence

CFG : Control Flow Dependence

PDG : Program Dependence Graph

SDG : System Dependence Graph

DDG : Dynamic Dependence Graph

DDSG : Dynamic Dependence Summery Graph

d-u : Defination/Use

Chapter-1
Introduction

1.1 Introduction

Software maintenance is major part of software development process. From complete life time of software system, 60% of the cost is spend on software maintenance. In software maintenance programmer spend half of time in program understanding the existing software, so Mark Weiser 1979 introduced program slicing technique to reduce cost of software maintenance and time required by programmers to understand the software. Program slicing can simplify the program by removing irrelevant part of the program with respect to particular slicing criteria.

Software debugging is the most important part of software testing. During software debugging main task is to find relevant statements that caused the failure. Hence program slicing allows the programmer to neglect the irrelevant statements and to focus on the relevant ones, so program slicing can be defined as deduction technique because in which start from the abstract program code to the concrete program run.

Many program slicing approach have been proposed in literature survey. These are divided into two parts static approach and dynamic approach. Static approach is based on statically available information and it introduced by Mark Weiser in 1979. This approach use dataflow equations to find statement relevant to a set of variables at the point of interest. Dynamic slicing was introduced by B. Korel and Laski in 1988 to find the statements at execution time based on particular input.

Using control flow and data flow dependences one can check for specific defect patterns and focus on specific subsets of the program being debugged (the subset that may have influenced a specific statement or the subset that may be influenced by a specific statement).Such a subset is called a slice, and the corresponding operation is called slicing. [3]

1.2 Background of Program Slicing

Mark Weiser introduced program slicing concept. “Program slicing is an automated means of determing possible origins of a variable value”. Using program slicing, one can effectively narrow down the number of possible infection sites. [3]

In computer programming, program slicing is the computation of the set of programs statements, the program slice that may affect the values at some point of interest, referred to as a slicing criterion. Program slicing can be used in debugging to locate source of errors more easily. Other applications of slicing include software maintenance, optimization, program analysis, and information flow control.

1.3 Fundamental Concept

In this Section the basic concepts and terminologies related to thesis work has been discussed and that are used in next chapter. And also few details about representation of a program in various form.

1.3.1 Program Slicing

According to Mark Wesier in [1], "Program slicing is program analysis technique that uses program statement dependencies information to identify parts of a program that influence or are influenced by an initial set of program points of interest which is called the slice criteria".

Using program slicing, one can effectively narrow down the number of possible infection sites. [3]

1.3.2 Slicing Criteria

The pair $< s, v >$ is known as Slicing Criterion where 's' is a program point of interest and 'v' is a variable used or defined in s. [1]

Slicing is classified into three different categories.

- Type of Slicing
- Direction of Slicing
- Level of Slicing

1.3.3 Type of Slicing

Slicing is divided into two types according to run time environment.

- Static Slicing
- Dynamic Slicing

1.3.3.1 Static Slicing

A static program slice consists of all statements in program P that may affect the value of variable v at some point s. The slice is defined for a slicing criterion $C = (s, v)$ where s is a statement in program P and v is a subset of variables in P. A static slice includes all the statements that affect variable v for a set of all possible inputs at the point of interest (i.e., at the statement s). Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data flow and control flow dependencies. Static slice contains all statements that may affect a variable for every possible execution. [7]

Static slicing is basically used to examining the source code without actually executing the program and also helps in understanding the program. But not used in understanding of program execution. Static slicing contains relatively large slices as compared to dynamic slice.

Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies.

Example-

```
1.  void main()
2.  {
3.  int i;
4.  int sum = 0;
5.  int product = 1;
6.  for(i = 0; i < N; ++i) {
7.  sum = sum + i;
8.  product = product *i;
9.  }
10. cout<<sum;
11. cout<<product;
12. }
```

Figure-1.1 (sample program for introducing Static Slicing)

A valid slicing of the above program (In Figure-1.1) with respect to the criteria $(cout(sum), \{sum\})$

```
1.  void main()
2.  {
3.  int i;
4.  int sum = 0;
5.  int product = 1;
6.  for(i = 0; i < N; ++i) {
7.  sum = sum + i;
8.  product = product *i;
9.  }
10. cout<<sum;
11. cout<<product;
12. }
```

Figure-1.2(Static Slice of above program in Figure-1.1)

1.3.3.2 Dynamic Slicing

Korel and Lasky proposed dynamic slicing which used dynamic analysis to identify only the statements that affect the variables at point of interest at the particular anomalous execution trace. In this way, the size of the slice can be considerably reduced. [4][5]

Dynamic slice contains all statements that actually affect the value of a variable at a program point of interest for a particular execution of the program rather than all statements that may have affected the value of a variable at a program point of interest for any arbitrary execution of the program. [6]

Another advantage of dynamic slicing is the run-time handling of arrays and pointer variables.

Dynamic slicing gives small slice in comparison of static slicing. Dynamic slicing is mostly used to understand program execution. Programmers may still have difficulties to understand the program and its behavior. The slicing tools usually developed provide limited support during the process of understanding of large programs and their executions. Therefore, it is important to develop methods that will support the process of understanding of large software systems. One aid to understanding of large software systems is to use an intermediate representation of a program and then compute a slice from the graph. This slicing technique aims to give a better

understanding of large programs and their executions for a particular input. Static program slicing and dynamic program slicing when combined together with different program slicing visualization tools, helps in better understanding of large programs and their executions. [6]

So that dynamic program slice is useful in program fault localization, software testing and software maintenance, reverse engineering, software debugging, program understanding.

Example:-

```
1.  void main()
2.  {
3.  int n,a,b,c;
4.  printf("Enter values");
5.  scanf("%d%d%d",&n,&a,&b);
6.  if(n>0)
7.  {
8.  if(a>b)
9.  c=a;
10. else
11. c=b;
12. }
13. else
14. c=n;
15. printf("%d",c);
16. }
```

Figure-1.3(Sample program for introducing Dynamic Slicing)

Dynamic slice for above program (In Figure-1.3) with respect to slice criteria($printf(\text{“}\%d\text{”}, c), \{c\}$) and inputs are $n = -4$, $a = 10$, $b = 8$.

```
1.  void main()
2.  {
3.  int n,a,b,c;
4.  printf("Enter values");
5.  scanf("%d%d%d",&n,&a,&b);
6.  if(n>0)
7.  {
8.  if(a>b)
9.  c=a;
10. else
11. c=b;
12. }
13. else
14. c=n;
15. printf("%d",c);
16. }
```

Figure-1.4(Dynamic Slice of above Program in Figure-1.3)

1.3.4 Direction of Slicing

According to traversal, Slices are divided into two parts.

- Backward Slicing
- Forward Slicing

1.3.4.1 Backward Slicing

A backward slice of a program with respect to a program point s and set of program variables v consists of all statements and predicates in the program that may affect the value of variables in v at s.

i.e., Backward slice of B, one proceeds backward along the dependences. Thus, all statements that could have influenced B can be determined. This is most useful in determining where the program state at execution of B could have come from. [3]

Formally, the backward slice $S^B(B)$ is computed as

$$S^B(B) = \{A | A \rightarrow * \; B\}$$

B is called the slicing criterion of $S^B(B)$

Example:-

1. Class xyz
2. {
3. int incr(int a,int b)
4. {
5. return(a+b);
6. }
7. };
8. int main()
9. {
10. int i=1;
11. int sum=0;
12. xyz obj;
13. while(i<21){
14. sum= sum + i;
15. i=obj.incr(i,1); }
16. cout<<"Sum ="<<sum;
17. **cout<<"i = "<<i; (this is slicing criteria)**
18. }

Figure-1.5(Sample program)

1. Class xyz
2. {
3. **int incr(int a,int b)**
4. **{**
5. **return(a+b);**
6. **}**
7. };
8. **int main()**
9. **{**
10. **int i=1;**
11. int sum=0;
12. **xyz obj;**
13. **while(i<21){**
14. sum= sum + i;
15. **i=obj.incr(i,1);**}
16. cout<<"Sum ="<<sum;
17. **cout<<"i = "<<i;**
18. }

Figure-1.6 (Backward Slice of Sample Program in Figure-1.5)

Backward slice of above program (In Figure-1.5) with respect to slicing criteria $C(cout << "i = " << i, \{i\})$

1.3.4.2 Forward Slicing

A forward slice of a program with respect to a program point s and set of program variables $\boldsymbol{v}$ consists of all statements and predicates in the program that may be affected by the value of variables in $\boldsymbol{v}$ at s.

More specifically the forward slice of $S^F(A)$. Formally, it consists of all statements that (transitively) depend on A. [3]

$$S^F(A) = \{B | A \rightarrow +B\}$$

In a slice S^F (A) the originating statement *A* is called the slicing criterion.

Example:-

Forward Slice of above program (In Figure-1.5) with respect to slicing criteria $(int\ sum = 0, \{sum\})$

```
1.  Class xyz
2.  {
3.      int incr(int a,int b){
4.        return(a+b); }
5.  };
6.  int main()
7.  {
8.    int i=1;
9.    int sum=0;  (This is slicing criteria)
10.   xyz obj;
11.   while(i<21){
12.     sum= sum + i;
13.     i=obj.incr(i,1); }
14.   cout<<"Sum ="<<sum;
15.   cout<<"i = "<<i;
16. }
```

Figure-1.7 (Forward Slice of Sample Program in Figure-1.5)

1.3.5 Level of slices

There are two levels of slices.

- Intraprocedural slice
- Interprocedural slice

1.3.5.1 Intraprocedural slice

When find slice within one procedure then this is called Intraprocedural slice .it is not useful where any program use function call or procedural call. In which can't find slice over entire program.

1.3.5.2 Interprocedural slice

When compute the slice over entire program where program use different function call then this is called Interprocedural slice.

There are two ways to find Interprocedural slice.

- **UP –** In this method follow path from sliced procedure to calling procedure.
- **DOWN -** In this method follow path from sliced procedure to called procedure.

1.3.6 Execution Trace

It is the sequence of executed program for a particular input. Execution trace is typically used for dynamic backward slicing to find dynamic program slice.

1.3.7 Variable Slice

A set of relevant program statements in a program, In which value of variable affected directly and indirectly at point of interest.

1.3.8 Expression Slice

An expression contains variables, function calls, operators. i.e., expression slice is set of slices of those variables and function calls.

1.3.9 Statement Slice

A statement contains expressions. That is statement slice is set of slices of those expressions.

1.3.10 Slicing Operations

Slice include following operations.[3]

- **Chop** - The intersection between a forward and a backward slice is called a chop. Chop is useful for finding out how some statement A (originating the forward slice) influences another statement B (originating the backward slice).
- **Backbone -** The intersection between two slices is called a backbone slice, or backbone for short. A backbone is useful for finding out those parts of an application that contribute to the computation of several values.
  In debugging, finding a backbone is most useful if one has multiple infected values at different places and wants to determine a possible common origin.
- **Dice** - Difference between two slices is called dice. Dice is very useful to find out how the backward slice of some variable differs from the backward slice of some other variable.

Dice is most useful if one knows that a program is largely correct i.e., dice is most useful to detect small error in an almost correct large programs. By subtracting the backward slices of the correct variables from the backward slices of the infected variables one can focus on those statements that only contribute to the infected value

```
1.  int main(){
2.    int a,b,sum.mul;
3.    sum=0;
4.    mul=1;
5.    a=read();
6.    b=read();
7.    while(a<=b){
8.    sum = sum + a;
9.    mul = mul * a;
10.   a = a + 1;
11.   }
12. Write(sum);
13. Write(mul);
14. }
```

(a)

```
1.  int main(){
2.
3.
4.  mul=1;
5.  a=read();
6.  b=read();
7.  while(a<=b){
8.
9.  mul = mul * a;
10. a = a + 1;
11. }
12.
13. Write(mul);
14. }
```

(b)

```
1.  int main(){
2.
3.
4.
5.  a=read();
6.  b=read();
7.  while(a<=b){
8.
9.
10. a = a + 1;
11.    }
12.
13.
14. }
```

(c)

```
1.  int main(){
2.
3.
4.  mul=1;
5.
6.
7.
8.
9.  mul = mul * a;
10.
11.
12.
13. Write(mul);
14. }
```

(d)

Figure -1.8 (In this figure slice operations can shown (a) entire program (b) Backward slice S^B(13 or write(mul)) (c) Backbone of $S^B(12) \cap S^B(13)$ (d) Dice of $S^B(12) \times S^B(13)$) [3]

1.3.11 Dependence

Statement dependence is of two types.

- Data Dependence
- Control Dependence

1.3.11.1 Data Dependence

A statement B is data dependent on a statement A if

- A writes some variable v that is being read by B.
- There is at least one path in the control flow graph from A to B in which v is not being written by some other statement.

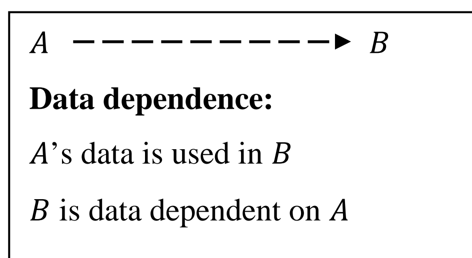
In other words, the outcome of A can influence the data read by B.

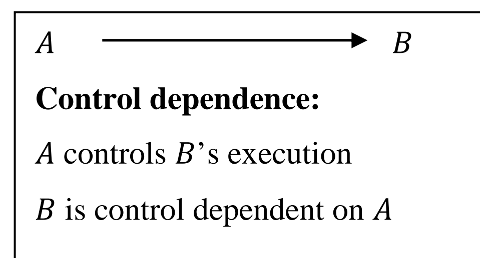
Figure 1.10 shows the data dependences in the fib() program. By following the dashed arrows on the right side, one can determine where the data being written by some statement are being read by another statement. For instance, the variablef0 being written by statement 2,f0 = 1, is read in statement 6,f = f0 + f1.

1.3.11.2 Control Dependence

A statement B is control dependent on a statement A if B's execution is potentially controlled by A. In other words, the outcome of A determines whether B is executed.

The solid arrows on the left side of Figure 1.10 show the control dependences in fib(). Each statement of the body of the while loop is dependent on entering the loop. All other statements are dependent on the entry of the function.





Example-

fibo.c -- Fibonacci C program to be debugged

```
1.  #include <stdio.h>
2.  int fib(int n)
3.  {
4.  int f, f0 = 1, f1 = 1;
5.  while (n > 1)
6.  {
7.  n = n -1;
8.  f = f0 + f1;
9.  f0 = f1;
10. f1 = f;
11. }
12. return f;
13. }
14. int main()
15. {
16. int n = 9;
17. while (n > 0)
18. {
19. printf("fib(%d)=%dN", n, fib(n));
20. n = n -1;
21. }
22. return 0;
23. }
```

Figure-1.9 (Sample Program)

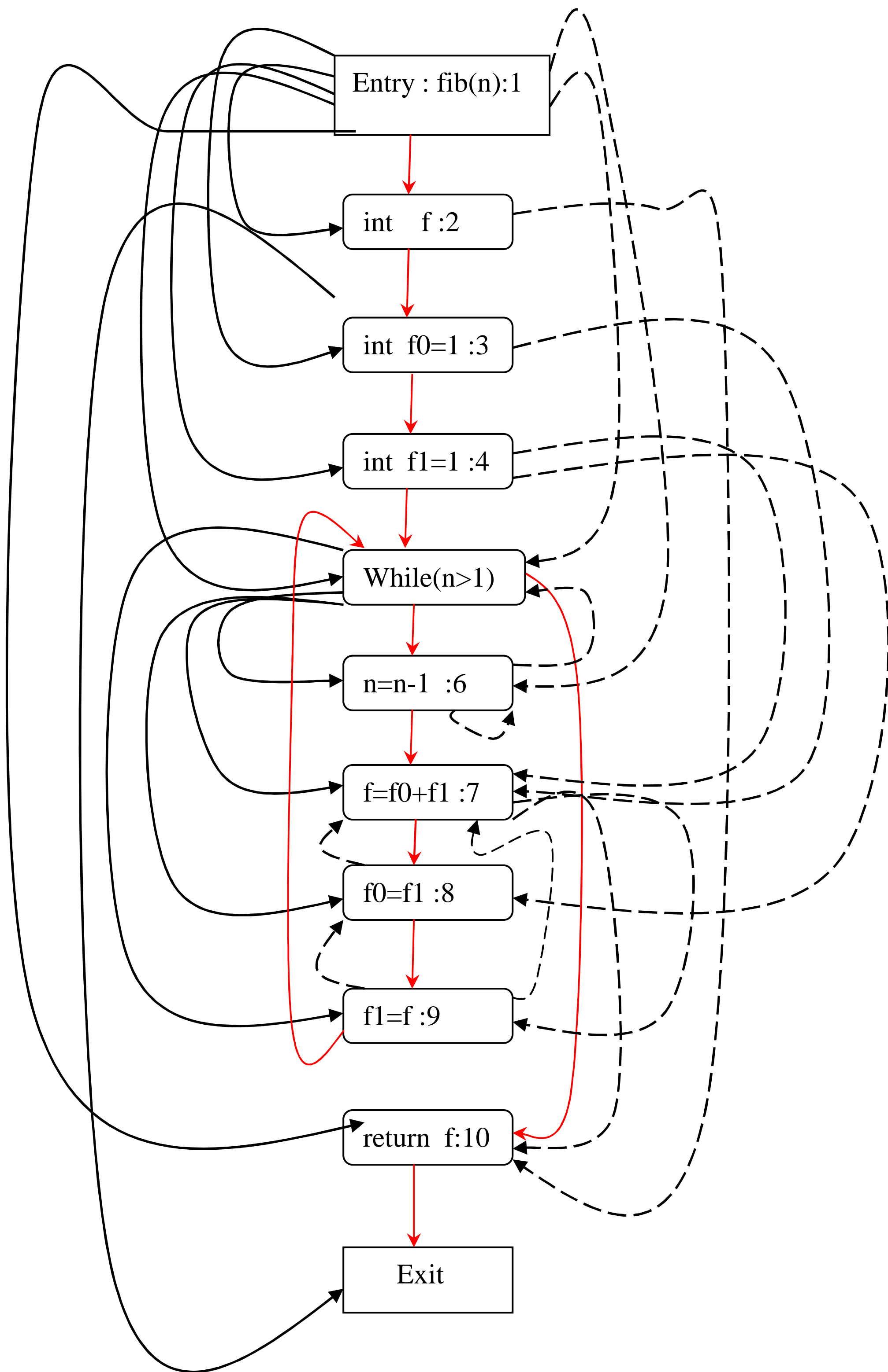


Figure-1.10 (Dependence graph of Sample Program in Figure-1.9)

1.4 Tools of Program Slicing

- **FINDBUGS -** FINDBUGS tool was developed by Hovemeyer and Pugh. It is open source. Its project page is found at http://findbugs.sourceforge.net/ . [3]
- **CODESURFER -** CODESURFER is considered the most advanced static analysis tools among all available tools. It is available free of charge to faculty members (if you are a student, ask your advisor). All others must purchase a license. CODESURFER is available at http://www.codesurfer.com/. [3]

  CodeSurfer is the tool of choice for organizations that manually review software for critical applications. It is developed by GrammaTech and it is also called GrammaTech's automated source-code analysis tool that finds bugs. CodeSurfer is a program-understanding tool that makes manual review of code easier and faster. Many program-understanding tools interpret code loosely. In contrast, CodeSurfer does a precise analysis. Program constructs including preprocessor directives, macros, and C++ templates are analyzed correctly.

  Forward and backward slice computation of any program using CodeSurfer is shown in these snapshots.

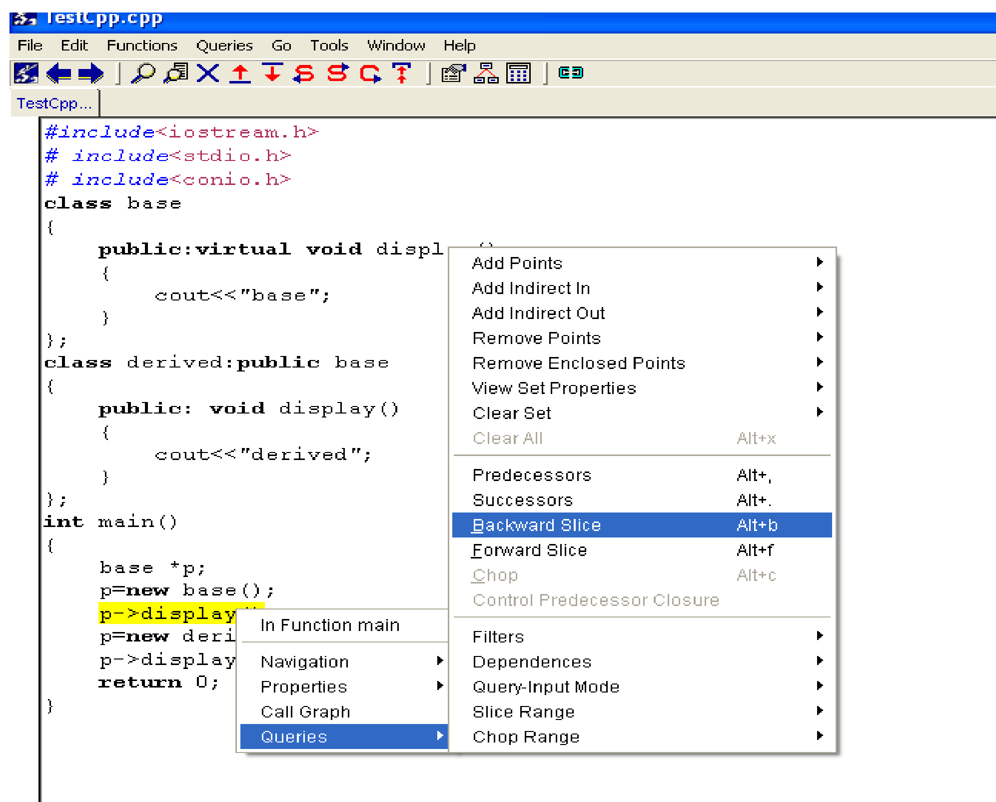


Figure-1.11

TestCpp.cpp

File Edit Functions Queries Go Tools Window Help

TestCpp...

```
#include<iostream.h>
# include<stdio.h>
# include<conio.h>
class base
{
	public:virtual void display()
	{
		cout<<"base";
	}
};
class derived:public base
{
	public: void display()
	{
		cout<<"derived";
	}
};
int main()
{
	base *p;
	p=new base();
	p->display();
	p=new derived();
	p->display();
	return 0;
}
```

Figure-1.12

TestCpp.cpp

File Edit Functions Queries Go Tools Window Help

TestCpp...

```
#include<iostream.h>
# include<stdio.h>
# include<conio.h>
class base
{
	public:virtual void di
	{
		cout<<"base";
	}
};
class derived:public base
{
	public: void display()
	{
		cout<<"derived";
	}
};
int main()
{
	base *p;
	p=new base();
	p->disp
	p=new d
	p->disp
	return
}
```

In Function main
Navigation
Properties
Call Graph
Queries

Add Points
Add Indirect In
Add Indirect Out
Remove Points
Remove Enclosed Points
View Set Properties
Clear Set
Clear All Alt+x
Predecessors Alt+,
Successors Alt+.
Backward Slice Alt+b
Forward Slice Alt+f
Chop Alt+c
Control Predecessor Closure
Filters
Dependences
Query-Input Mode
Slice Range
Chop Range

Figure-1.13

TestCpp.cpp

File Edit Functions Queries Go Tools Window Help

TestCpp...

```
#include<iostream.h>
# include<stdio.h>
# include<conio.h>
class base
{
     public:virtual void display()
     {
          cout<<"base";
     }
};
class derived:public base
{
     public:  void display()
     {
          cout<<"derived";
     }
};
int main()
{
     base *p;
     p=new base();
     p->display();
     p=new derived();
     p->display();
     return 0;
}
```

Figure-1.14

Figure-1.11, 1.12, 1.13, 1.14 show backward and forward slice with respect to slice criteria. In these snapshots first select any statement of given program then right click then select queries option in open window after select forward or backward option .then CodeSurfer high- lighted those statements which are affected by that statement or which are affected that statement. In these snapshots select p-> display() statement and to find backward and forward slice of this program

1.5 Difference Between Static Slice and Dynamic Slice

Consider a small piece of a program unit. which has an iteration block containing an if-else block. There are a few statements in both if and else blocks that have an effect on a variable. In case of static slicing focus whole program as a unit irrespective of the values of variables at execution and flow of control in if-else blocks, the affected statements in both blocks would be included in the slice. But, in case of dynamic slicing consider a particular execution of the program, where if block gets executed and the affected statements in the else block do not get executed. So, in this particular execution case, the dynamic slice would contain only the statements in if block.

Let's consider in figure-1.15 and slicing criteria $< 9, \{a\} >$. Static slicing of given program with respect to slice criteria includes statement number $\{1, 4, 6, 8\}$. But in

dynamic slicing for this criterion includes statement number {1, 4, 6} or {1, 4, 8} depending upon value of i. because execution of if block and else block depend upon value of i.

```
1.  void main()
2.  {
3.  int i, a, c;
4.  c=10;
5.  if(i>0)
6.  a=c;
7.  else
8.  a=c;
9.  cout<<a;
10. }
```

Figure -1.15 (Sample Program)

1.6 Application of Program Slicing

Program slicing is used in various area like software maintenance, program differencing, program integration, software testing, debugging, reverse engineering.

- **Program Differencing: -** In process of program development new changed components are tested. Thus program differencing identifies new change syntactically and semantically is useful. [30] Program slicing can be applied in program differencing, by computing and comparing slices and their behaviors, before and after performing modification.
- **Program Integration:-** program integration [30] is the process to merge two program A and B. Program integration relies on program slicing to produce slices with respect to affected point and compare them to determine if A and B can be integrated with each other.
- **Software Maintenance:**- In software maintenance when modify in one part of the software which has ripple effect on another part of software. Backward and Forward program slicing is very useful in ripple effect analysis.
- **Software Testing:-** data flow testing can be applied to check all d-u pairs. Program slicing is very useful in constructing test case, that tests all d-u pairs. Program slicing is also used in regression testing and incremental testing.

- **Debugging: -** debugging is the process to find the bugs and remove the bugs that lead to an unexpected behavior. Backward slice can be used to extract those statements that affect the variable at the point where the wrong value produced. Dynamic slicing is used to semi automated debugging of any program.[4]

1.7 Structure of the Thesis

The rest of the thesis is organized in the following order:

Chapter 2 In this chapter describes the related Work.

Chapter 3 In this chapter describes gaps of previous approach and related question to problem statement.

Chapter 4 In this chapter describes our proposed pseudo code for forward static slicing and its Implementation.

Chapter 5 explains the experiments performed and evaluates the results achieved

Chapter 6 In chapter conclude the thesis and discuss the future work that can be done in this area.

Chapter-2

Literature Review

2.1 Static Slicing Algorithms

Mark Weiser [1] was first introduced static program slicing "the process of identifying all relevant statements that affect the value of variable v at point of interest s in program P". In order to identify all relevant and irrelevant statements in program P with respect to particular slicing criteria $C = (s, v)$.

Static slicing algorithms are divided into two parts PDG based algorithms and SDG based algorithms. In next section illustrate major difference between in PDG and SDG based algorithms.

2.1.1 PDG Based Algorithms

Ottenstein [31] first defined program dependence graph (PDG) and then Horwitz et al [30] redefine the original PDG. Program Dependence Graph (PDG) consist of node and directed edge. It is used to track control flow and data flow in any program. Each edge in graph represents the reachable relationship from one statement to another statement (Control dependence) and from one definition of a variable to its usage (Data dependence).

Any simple program contains simple and compound statement. Simple statement includes assignment, read, write statements. Compound statement include conditional and loop statements. it is use for compute slice of a program . This graph is the basis for a number of program-analysis techniques, as it reflects all influences within a program. [8]

Consider a sample program from Figure- 2.1 to illustrate control and data dependence. In this program statements 8 and 9 are control dependent on statement 6(predicate test block) and variable sum in statements 4 has data dependence on statement 8 and 11, where variable sum define first time.

```
1.  Void main()
2.  {
3.  Int  i, sum;
4.  sum=0;
5.  i=1;
6.  while(i<=10)
7.  {
8.  sum=sum + i;
9.  i++;
10. }
11. Cout<<sum;
12. Cout<<i;
13. }
```

Figure-2.1 (Sample program for PDG)

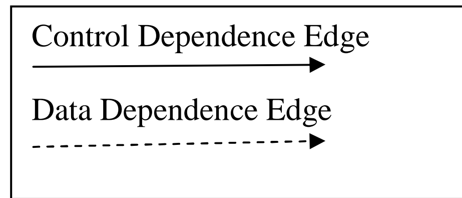


Figure-2.2 (Symbols used in PDG)

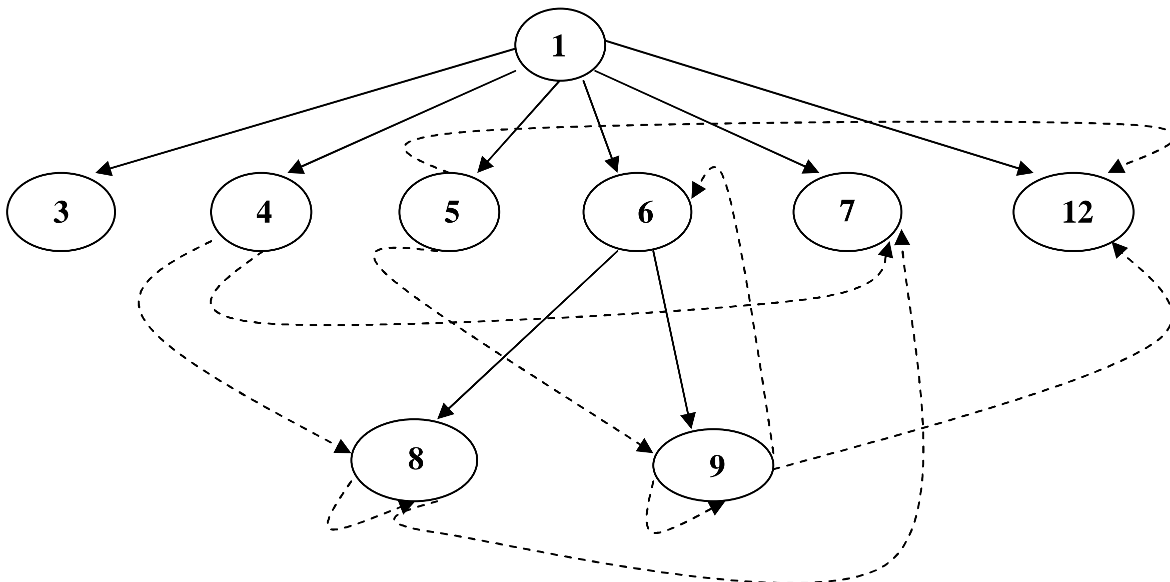


Figure-2.3 (Program Dependence Graph of Sample Program in Figure-2.1)

Computing Static slicing using PDG:-

In the first step construct PDG of program. All directly reachable statements from an executable point of program will be included and marked if either data or control dependence exist. All statements and relationships between statements are considered. each node *S* within a PDG represents a program statement, and outgoing edge of *S* represents data dependence or control dependence other nodes have upon *S*.

In the second step within PDG static slice for a variable v at node S can be computed as follow:-

Traverses backward all reachable edge of PDG starting from node S. this process iterate until all reachable nodes are visited and included into slice of S.

For example, to compute static slice with respect to slice criteria $C(12, i)$ in the sample program shown in Figure–2.1 algorithm traverse backward in PDG shown in Figure-2.3 from node 12 then found nodes are 9,5,6 and 1 that are the part of static slice for (12,i). In figure-2.4 color node show static slice with respect to given slice criteria.

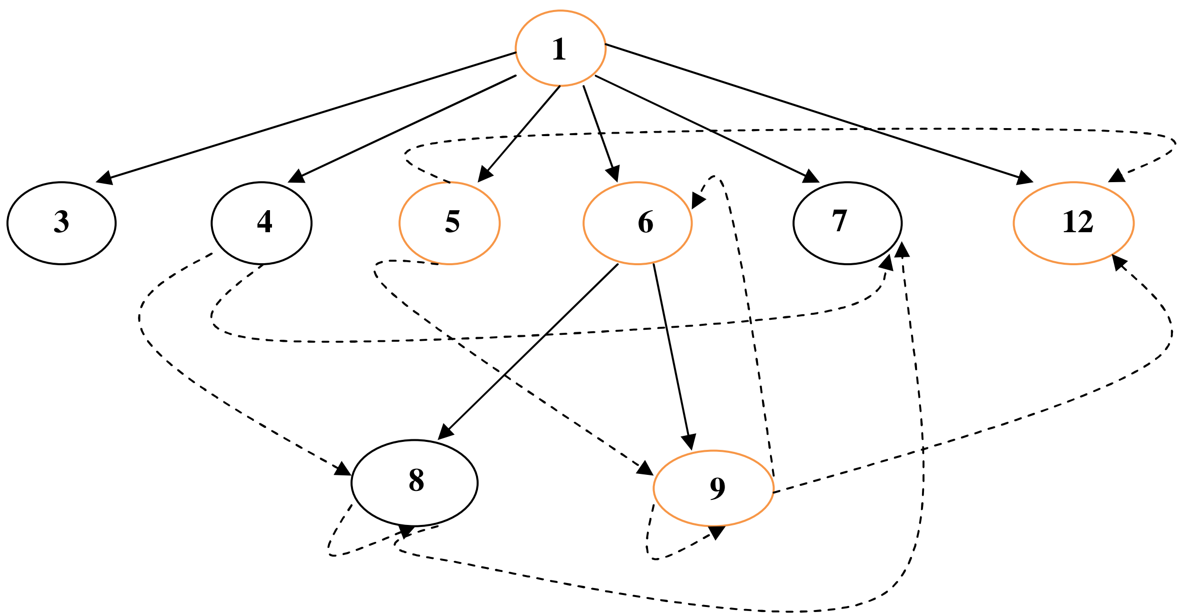


Figure-2.4 (Static slice for slice criteria $(12, i)$ using PDG)

2.1.2 SDG Based Algorithms

A SDG is a collection of procedure dependence graphs, one for every individual procedure. A procedure dependence graph represents a procedure in which statements of the procedure are shown by vertices. In a SDG, there are different kinds of edges are used. They can be classified as control dependence edges, data dependence edges, parameter edges, summary edges etc. [9]

The PDG cannot handle procedure calls, so Horwitz was introduced the System Dependence Graph (SDG) representation, which models the main program together with all associated procedures. SDG is actually a collection of PDGs. For programs without procedure calls, the PDGs and SDGs are similar. For construction of an SDG,

first the PDGs of all the procedures are constructed individually and then the SDG is constructed by integrating all the PDGs.
SDG is use to find Interprocedural slicing of any program. Because PDG is not useful to find the slice of any program which has multiple procedure call. [10]
A procedure dependence graph represents a procedure as a graph in which vertices are statements or predicate expressions. Data dependence edges represent flow of data between statements or expressions and control dependence edges represent control conditions on which the execution of a statement or expression depends. Each procedure dependence graph contains an entry vertex that represents entry into the procedure. To model parameter passing, an SDG associates each procedure entry vertex with formal-parameter vertices a formal-in vertex for each formal parameter of the procedure and a formal-out vertex for each formal parameter that may be modified by the procedure. An SDG associates each callsite in a procedure with a call vertex and a set of actual-parameter vertices an actual-in vertex for each actual parameter at the callsite and an actual-out vertex for each actual parameter that may be modified by the called procedure. [13]

SDG has different type of node to represent procedure calls and parameter passing. [11] it include like

- Procedure call statements are represented by call site nodes in the program.
- Actual-in and Actual-out nodes represent the input and output parameters at call site. They are control dependent on the call-site nodes.
- Formal-in and Formal-out nodes represent the input and output parameters at called procedures. They are control dependent on procedure's entry node.

SDG has different type of edges. These are used to link different type of node in system dependence graph. [11]

- Call edges link the call-site nodes with the procedure entry nodes.
- Parameter-in edges link the actual-in nodes with the formal-in nodes.
- Parameter-out edges link the formal-out nodes with the actual-out nodes.
- Summary edges are added to represent the transitive dependencies that arise due to procedure calls.

```
3. Class xyz {
    Protocted :
    int a,b,c;
    int sum,i;
    Public:
4.  int  add(int a , int b){
5.  c=a+b;
    return(c); }
6.  int decrement( int a) {
7.   a=a-1;
    return(a);
    }
8.  void display(){
9.   sum=0;
10. i=10;
11. while(i>0){
12. sum=add(sum,i);
13. i=decrement(i);
      }
14. cout<<"sum ="<<sum;
      }
1.   void main()
     {
      xyz   obj ;
2.   obj.display();
      }
```

Figure-2.5 (Sample program introducing SDG)

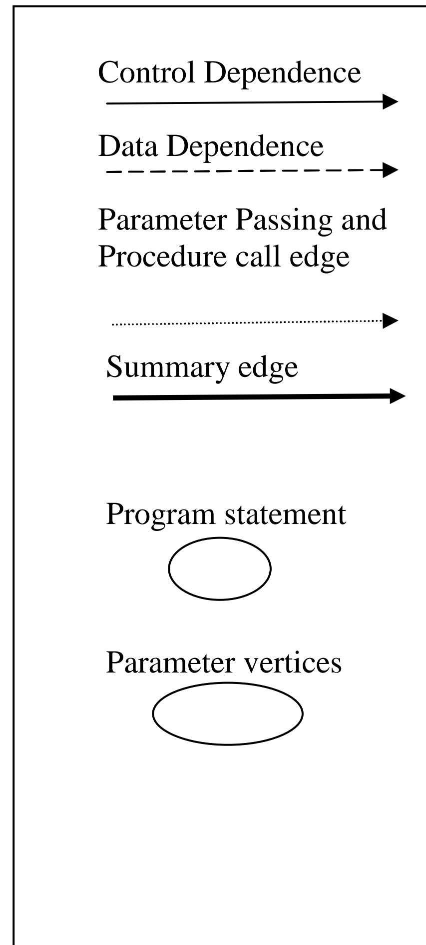


Figure-2.6 (Symbols Used in SDG)

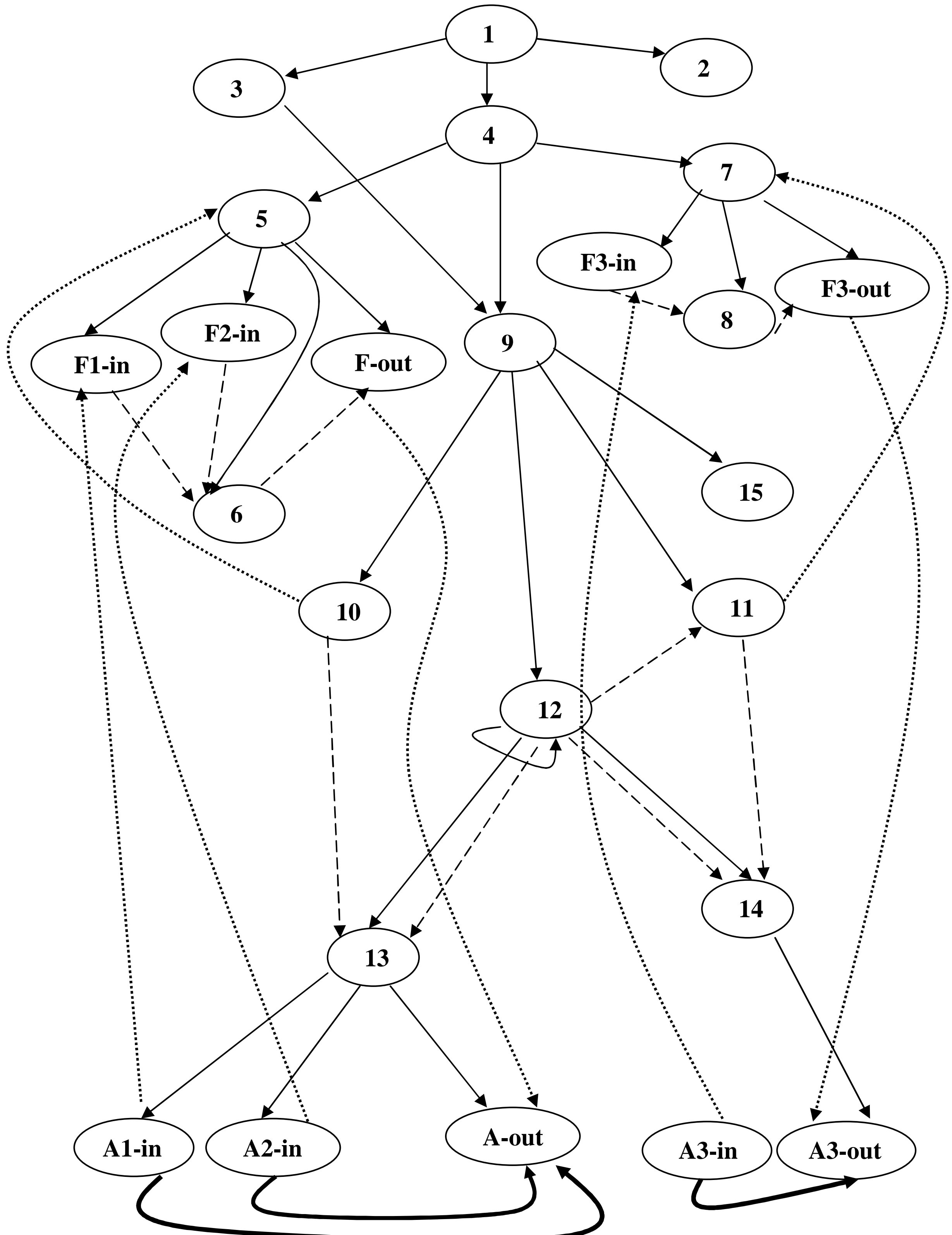


Figure-2.7 (SDG of Sample program in Figure -2.5)

2.2 Dynamic Slicing

Korel and Laski [4][5] introduced dynamic slicing that consider only one path of program execution rather than all path of program execution**.**

A dynamic slice is executable subset of a program, for variable at specific position. A slicing criteria of a program P executed on input x is $C(x, v^s)$ where v is a variable at execution position at s. [4]

Agrawal and Horgan also proposed definition of dynamic slice. Dynamic slice of a variable is set of all statement in execution history whose execution affect on value of variable. Main difference Between Korel's algorithm and Agrawal and Horgan's algorithm is that former find non-executable slice but after some time emphasize on executable slice.

Dynamic slice algorithm divided into Dynamic Backward Slicing and Dynamic Forward Slicing. Most of dynamic slicing algorithms are based on backward analysis [4] [5] [18]. In the backward analysis first record execution trace then dynamic slicing algorithm trace backward execution trace to find dynamic dependence to compute dynamic slice.

2.2.1 Dynamic Backward Slicing

Korel [4] introduced concept of Dynamic Backward Slicing. In this approach program executed at given set of inputs and record the execution trace. For a variable v at point of interest S. dynamic backward algorithms trace back recorded execution trace to find data and control dependencies that are useful in computation of v at s.

Korel's Dynamic Backward Algorithm –[4]

Input : slicing criteria $C = (x, v^s)$

T_x execution trace up to execution position s.

Output : dynamic slice of variable v at point s

1- Execute program P on input x and record execution trace T_x up to position s.
2- Set all nodes in T_X as unmarked and not visited.
3- Find last definition of v^P and set v^P as marked.
4- While there exist a marked and not visited action x^k in T_x do
5- Select marked but not visited action x^k in T_x
6- Set x^k is visited.
7- For all variable $v \in U(x^P)$ do find and mark last definition v^s of v.

8- Mark all action Z^t such that there exist a control dependence betwwen Z^t and x^k

9- Mark all multiple occurrence of node x.

10- End of while

11- Show the dynamic slice that is constructed from P by removing nodes(statements) whose action were not marked in T_x.

Korel algorithm first executes the program and record the whole execution trace T_x up to position s. Then initially all nodes in T_x are unmarked. Secondly according to slice criteria algorithm find last definition of v and set its status. then in loops find all data and control dependence.

Agrawal and Horgan's DDG (Dynamic Dependence Graph) Algorithm[17]

In this method Agrawal construct Dynamic Dependence Graph (DDG) from execution trace of the program before finding the slice. When building DDG a separate node is created for each occurrence of a statement in execution history with outgoing dependence edge to only those statements upon which this particular statement occurrence depends on**.**

After creating DDG the slicing algorithm check all data and control dependencies by traversing backwards in the graph.

```
1. read (n)
2. for I := 1 to n do
3. a := 2
4. if c1==1 then
5. if c2==1 then
6. a := 4
7. else
8. a := 6
9. z := a
10. write (z)
```

Figure-2.8 (Sample Program introducing DDG)

Suppose input value $n = 2\ and\ c1 = 1\ and\ c2 = 2$ then execution history is $\{1^1,2^1,3^1,4^1,5^1,6^1,9^1,2^2,10^1\}$

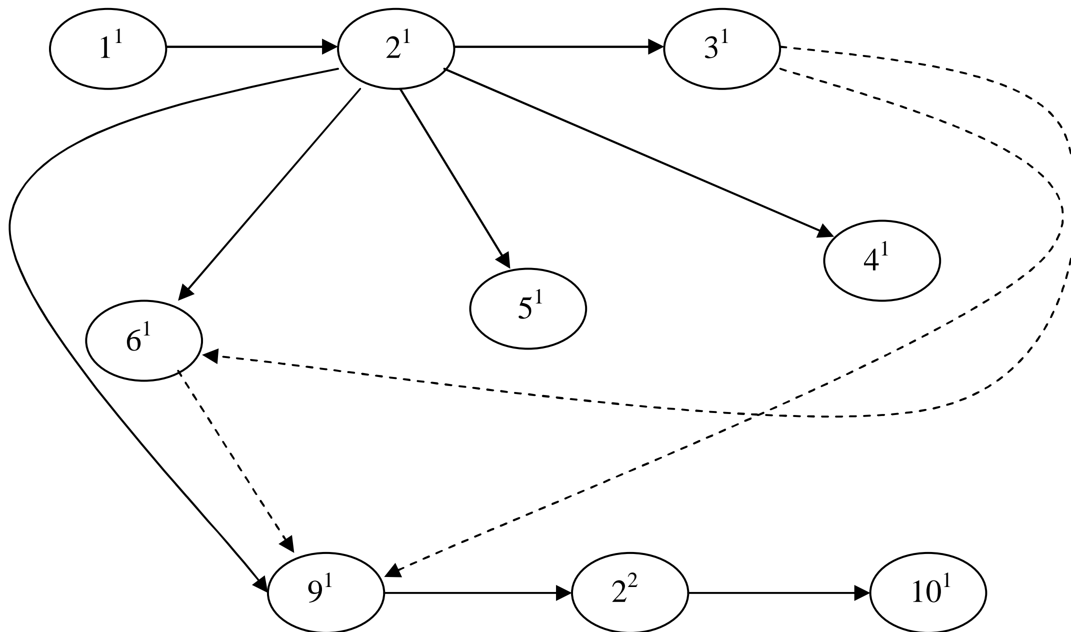


Figure-2.9 (Dynamic Dependence graph of sample program in Figure-2.8)

This algorithm doesn't compute executable slices and algorithm limited to structural programming language.

Kamkar's DDSG(Dynamic Dependence Summary Graph) Algorithm

Kamkar [32] presented a dynamic slicing method which is based on concept of graph reachability in dependence graph. During program execution a Dynamic Dependence Summary Graph is constructed which is similar to DDG presented by Agrawal. This algorithm requires execution to getting dependencies information. Kamkar's algorithm may also be used for slicing Interprocedural programs with existence of procedure calls and procedure. This algorithm doesn't compute correct slice for object oriented programs.

2.2.2 Dynamic Forward Slicing

Although Dynamic Backward Slicing provide many benefits in software debugging, program understanding, software maintenance, software testing etc. but in this approach entire execution trace recorded that have significant space and time consumption for computation of dynamic slicing.

Korel and Yalamanchili's Algorithm

In order to address the space complexity problem of the dynamic backward slicing

Approaches, Korel and Yalamanchili [20] introduced new algorithm of dynamic slices at run-time without the need of any major recording of execution traces.

According to Korel and Yalamanchili's algorithm, a structural program is set of blocks. Initially any block may be removed from source code. During dynamic forward computation, to identify these non-removable blocks that have to be include in program slice. Korel and Yalamanchili divided removable blocks into simple and complex blocks. Assignment and input and output and jump statements are as simple blocks. Predicate statements (if-else, while, for) are consider as complex blocks. Predicate statements are not allowed to be removed from a statement because it is not consider independent block. If Blocks are nested and if inner blocks are non removable then outer blocks automatically non-removable these include in slice.

Begin
read(N); B1
i:=1; B2
while (i<= N) do B3
read(X); B4
if(X<0)then B5
Y= getBig(X); B6
else
Y= getSmall(X); B7
end if;
Z=f3(Y) B8
Write(Z); B9
i=i+1; B10
End while;
End

Figure-2.10 (structure program marked with removable blocks)

In what follows use a sample program shown in Figure 2.10, to illustrate the notion of a block as introduced by Korel and Yalamanchili's algorithm. Blocks in the program

are individually identified by boxes and associated numbers. B1, B2, B4, B8, B9, and B10 are simple blocks. B3 and B5 are complex blocks. B4 is nested by B3 and B6, B7 are nested by B5.

An executed block B is not included in the slice of a variable $\boldsymbol{v}$ if any one of the following two conditions satisfied.

- B is a simple block and $\boldsymbol{v}$ is not defined (assigned) during the execution of B.
- B is a complex block, during the execution of B and all its nested blocks, variable v is not defined (assigned), and block B does not already belong to currently computed slice of v at run-time.

Since Yamanchili's forward slicing algorithm is a direct extension of Korel's forward algorithm, it will provide a more detailed description of Korel's algorithm. First, introduce the major data structures used by the algorithm.

- B is a block;
- BL is a stack of blocks containing the part of program being currently executed;
- BV(B) is a set of variables defined/modified during the current execution of block B;
- $Slice(v) = Slice(k, \{v\})$ is a dynamic slice of variable v at the current execution position k;
- NodeSlices(X) contains a union of slices of all used variables at current action X;
- TopSlice(B, v) is the copy of dynamic slice for variable v at the entry of block B;
- BlockFlag(B,v) is a tag for variable v as to block B, which is marked for each block B' in BL if currently executed block already exists in TopSlice(B,v).

Dynamic Forward Algorithm:

Input: A slicing criterion $\boldsymbol{C}\ (\boldsymbol{x}, \boldsymbol{n^s}, \{\boldsymbol{v}\})$

Output: A dynamic slice of variable v at position s

1. Execute program P on input x. On entry node do:

2. for all v Є V do SLICE (v)=Φ ;

3. for all x Є N do NODESLICE (X):={X};

4. Action X^k**:** On each action X^k the following steps are performed

5. 1- Action X^k

6. NODES LICE(X) **:=** NODESLICE(X) U SLICE(v), v Є U(X^k);

7. For all v Є Vdo

8. if v Є D(X^k) then

9. SLICE(v) **:=** NODESLICE(X);

10. for all B in BL do BV(B) := BV(B) U {v};

11. else

12. if X Є SLICE(v) or (X is no simple block) then

13. SLICE(v) **:=** SLICE(v) **U** NODESLICE(X);

14. endif

15. For all B in BL do

16. if X Є TopSlice(B,v) then

17. BlockFlag(B,v) **:** marked;

18. endfor

19. endfor

20. 2- Entry into block B

21.BL := Push B into BL

22.BV(B):=Ø;

23. For all vЄV do

24. TopS lice(B,v) **:** SLICE(v);

25. BlockFlag(B,v) **:=** unmarked;

26. endfor

27. 3-Exit from block B

28. BL:=Pop B off BL;

29. For all v Є V do

30. if(v Є BV(B)) and (BlockFlag(B,v) = unmarked) then

31**.** SLICE(v) := TopSlice(B,v);

32. endfor

33. 4- k = q (execution reaches position q);

34 Display SLICE (y)

Korel's algorithm starts with initializing all data structures, and then continues on computing program slices for all defined variables at run time. When execution arrives the entry of an action X in step 2, the undefined variables in current block and any unmarked variable corresponding to the current block make a copy of their own

slice as TopSilce. When slicing an action X in step 1, algorithm registers all defined/modified variables and computes/updates slice for each of them in step 1a. If current block B is a complex block, then B is temporarily included as slice to all variables except those currently defined in step 1b. Until execution arrives the exit of block B in step 3, this assumption may not be clarified true if B has been included into slices of v; otherwise, B is removed from slice of v . The algorithm terminates when the execution comes to an end s, which is a defined element in the slicing criteria $C\ (x, n^s, \{v\})$.

The method of Korel and Yalamanchili is based on removable blocks. Idea of this approach is that during program executed block should be include or not.

After that Tibor Gyimothy proposed a method for computing dynamic program slice based on D/U program representation. By using of the D/U program representation, after the last instruction has been executed then it can obtain the dynamic slice for all instruction executed previously. In which effect of data and control dependence can be treated in same way. So that doesn't need DDG. So that it can reduce space complexity. But it use execution trace to map D/U relationship to actual execution trace to find out dynamic slice.

In 2005 Zhao gave forward method of dynamic slicing on basis of Korel forward method. In which program coverage of predicate can express by relationship of domination between the blocks. But in this method it can produce some redundancy.

Chapter-3

Problem Statement

3.1 Research Question

- **Why Dynamic slicing ?**

  A slice of a program P is an executable part of P that computes the same function as P for variable v at some selected point of interest P. It consists of all statements and predicates of the program that might affect the value of v at point P. Actually, it is a kind of the static program slice.

  In the static program, it is required to find a program slice that involves all statements that may affect the value of a variable at a program point for any input set, it considered all the possible executable paths. So, it had a lot of redundancy.

  In the practical application, constraint on those statements that actually had an influence on some variable on the actual execution of program. Based on the requirement, Korel presented the concept of dynamic program slice for variable v at some point of interest s in the program P, when input x, the dynamic slice consisted of all statements that might affect the value of v at point s. It only considered someone path of program P, and it only included those statements that were involved in the actual execution of program with respect to a particular input. Actually, it is a subset of the static program slice, it can be considered a refinement of the static slice.[6]

  So that dynamic program slice has been shown to be useful in program fault localization, software testing and software maintenance.

- **Why Forward Dynamic Slicing?**

  - **Limitation of Backward Analysis Method of Dynamic Slicing**

    In Backward analysis size of dynamic dependence graph(DDG) is unbounded. So that space complexity is major problem. But in Forward Analysis space complexity is bounded.

  - **Limitation of Previous Forward Analysis Algorithms of Dynamic Slicing**

Till date the algorithms of dynamic program slice based on forward analysis are less. In the early 90, Korel and Yalamanchili introduced a forward method for determining dynamic program slices. Their algorithm computes executable program slices. In many cases these slices are less accurate than those computes by our forward dynamic slicing algorithm. (Executable dynamic slices may produce inaccurate results in the presence of loops. For the different definition of the same variable, it can produce inaccurate results.). The method of Korel and Yalamanchili is based on the notion of removable blocks. The idea of this approach is that during the program executed block should be included in the dynamic slice or not.

Then, Gyimothy proposed a method for computing dynamic program slice based on D/U program representation. By using of the D/U program representation, after the last instruction has been executed then it can obtain the dynamic slice for all the instructions processed previously. in this method, the effect of data and control dependence can be treated in the same way.

In 2005, on the basis of the forward algorithm by Korel presented, zhao introduced a forward approach of dynamic program slice. In this method, it can express the program's coverage of the predicate by the relationship of domination between the blocks, passing through the layers, the source program structure can be clearly shown As long as concerning the statement or statement block and the domination between the statement or statement block, it can compute the corresponding slices But in this method it can produce some redundancy (the variables which cannot be modified in complex block may be included).

3.2 Problem Statement

Develop a novel approach of Dynamic forward Slicing after analysis of previous forward dynamic slicing methods, which can reduce space complexity and improve precision.

Chapter-4

Proposed Algorithm of Dynamic Forward Slicing

4.1 Proposed Algorithm

Dynamic Forward Program slicing has lot of advantages like software testing and find variable dependence and program understanding and fault localization. Analyzing and reusing the code by the help of the program slices becomes easier. In larger programs, there is redundant and repeated use of the same code at execution time. But the programmer is unaware of the problem. By the use of this technique the redundancy is minimized. And by using forward dynamic slicing can minimize space complexity to a greater extent. Automatic differentiation of variables that are used for a particular statement of the program can be known easily. And this algorithm can find ,which variable used in which statement of program at execution time. So that it can find bugs easily in the programs.

Algorithm

Input :- A file containing C/C++ program and a variable (v) and line number (n) and conditional variable (c) and value of conditional variable (L).

Output:- A text file containing statements, which affected by variable(v) at value of conditional variable from line number(n) to end of program.

Process –1:

Step-1 First initialize variable list ($varlist$) =NULL

Step-2 Read variable (v) and line number (n) and conditional variable (c) and value of conditional variable (L) i.e., slicing criteria for dynamic slice.

Step-3 Find total number of line ($totalline$) in given program.

Step-4 Add variable (v) into variable list ($varlist$)

Step-5 while (number of variable in $varlist$) do

Step-6 $readline = n$

Step-7 call MainFunction ($readline, v, c, L$)

Step-8 end of while loop

Process-2:

Step-1 MainFunction ($readline, v, c, L$)

Step-2 while ($readline\ <=\ totalline$) do

Step-3 if (readline has conditional statement and conditional variable (c) is also present) Then

Step-4 call Function1 ($readline, v, c, L$)

Step-5 else if (readline has '$=$' operator and variable (v) is present in right hand side of '$=$' operator) Then

Step-6 call Function2 ($readline, v$)

Step-7 else if (readline has input/output function and variable(v) is also present)

Then

Step-8 call Function3 ($readline, v$)

Step-9 $readline = readline + 1$;

Step-10 end of while loop

Step-11 End of MainFunction

Process-3:

Step-1 Function1 ($readline, v, c, L$)

Step-2 if (at value L condition is true in readline)

Then

Step-3 call Mainfunction ($readline, v, c, L$) for true condition block

Step-4 else

Step-5 call MainFunction ($readline, v, c, L$) for false condition block

Step-6 End of Function1

Process-4:

Step-1 Function2 ($readline, v$)

Step-2 write readline into output file

Step-3 Add variable (v) which is present in left hand side of '$=$' operator into varlist.

Step-4 End of Function1

Process-5:

Step-1 Function3 ($readline, v$)

Step-2 write readline into output file

Step-3 End of Function3

4.2 Experimental Setup

- **Visual Studio 2008** Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It can be used to develop console and graphical user interface applications along with Windows Forms applications, web sites, web applications, and web services in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework and Microsoft Silverlight. Visual Studio supports different programming languages, which allow the code editor and debugger to support nearly any programming language. Built-in languages include C/C++, VB.NET, C#.
- **Language** C#
- **Platform** window vista

Chapter-5

Experimental Result

5.1 Sample program and Implementation

Consider sample program of C++ for taking as input file in our algorithm**.**

```
1.  void main()
2.  {
3.  int i,x,y,z,w=1,p;
4.  cout<<"Enter The vaue of i";
5.  cin>>i;
6.  while(i<10)
7.  {
8.  if(i==5)
9.  x=w;
10. else
11. y=w;
12. i++;
13. }
14. z=x+y;
15. p=z;
16. cout<<z;
17. cout<<p;
18. }
```

Figure-5.1(Sample Program)

Dynamic forward slice of above program (In Figure-5.1) with respect to Slicing criteria $C(4, w)$ and input variable is $i = 6$ given below:-

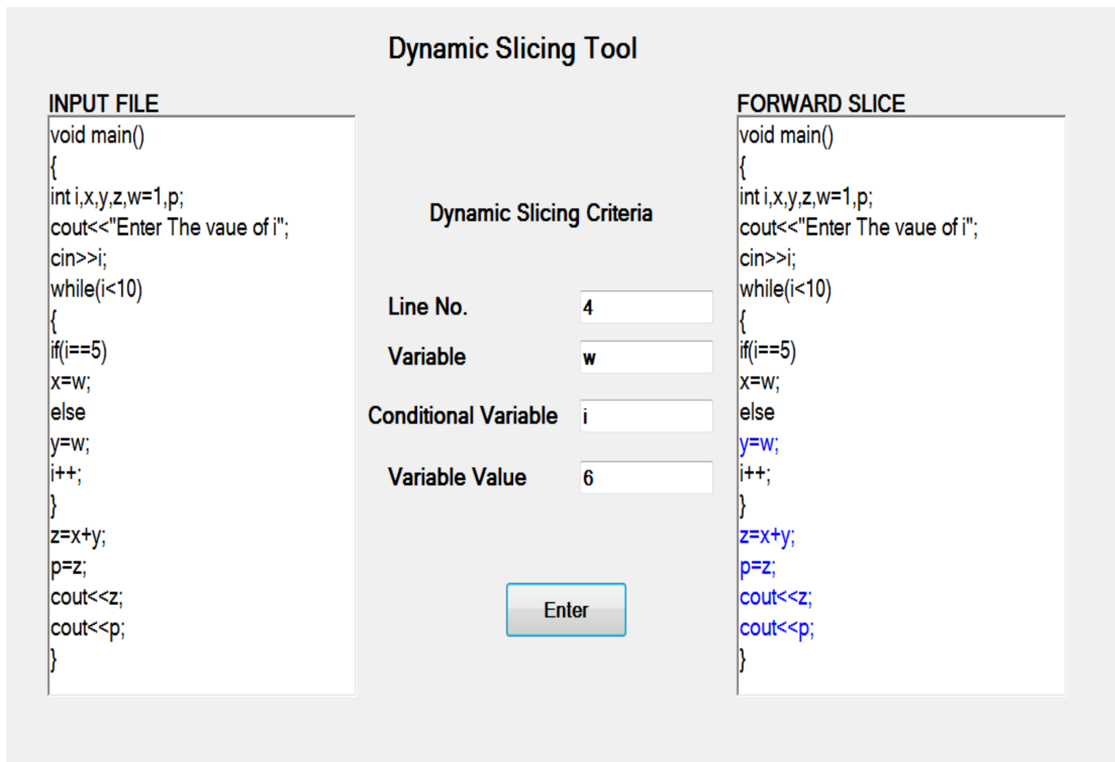


Figure-5.2

Dynamic forward slice of above program (in Figure-5.1) with respect to $C(4, w)$ and input variable $i = 5$ then

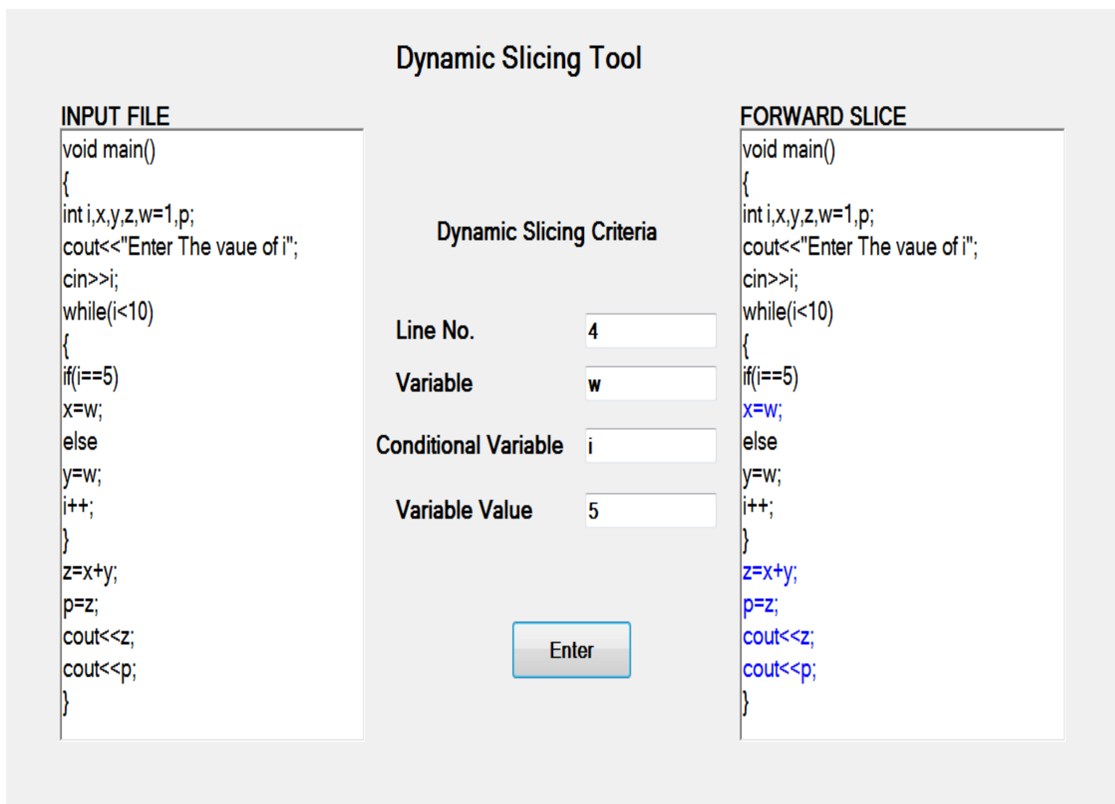


Figure-5.3

Dynamic forward slice of above program (in Figure-5.1) with respect to $C(4, w)$ and input variable $i = 12$ then

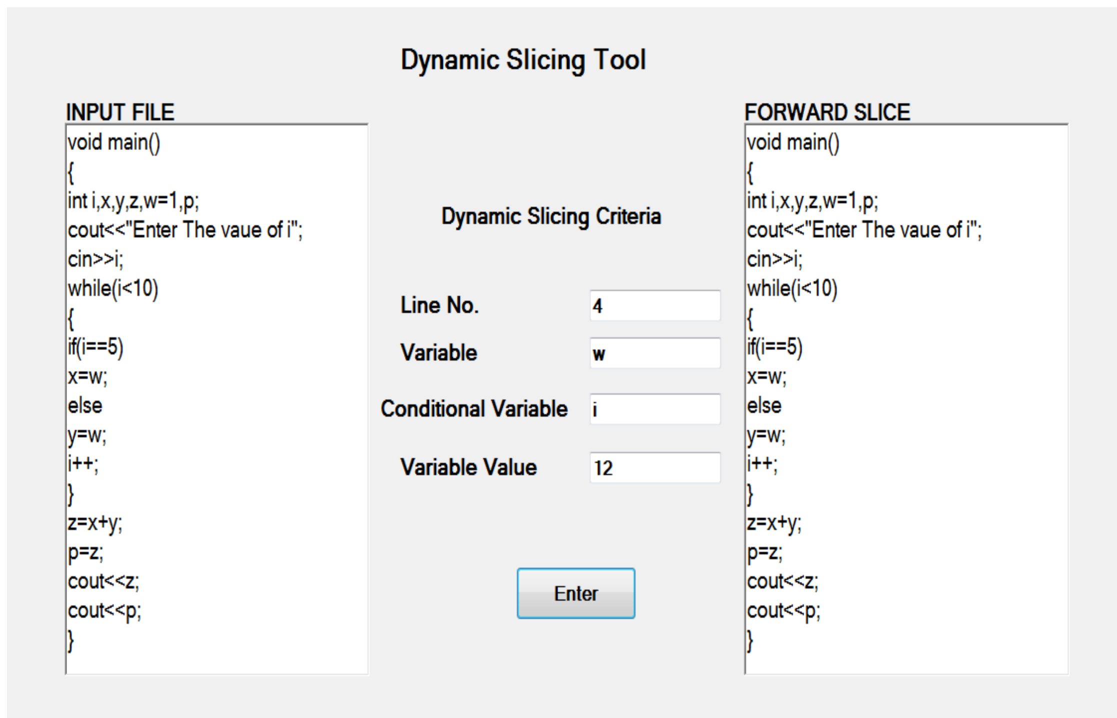


Figure-5.4

Another Example

```
1.  void main()
2.  {
3.  int i,a,b,c,d,e;
4.  cout<<"Enter the value of i";
5.  cin>>i;
6.  while(i<10)
7.  {
8.  a=c;
9.  b=a;
10. i++;
11. }
12. d=a;
13. e=d;
14. cout<<d;
15. cout<<e;
16. }
```

Figure-5.5 (Sample Program)

Dynamic forward slice of above program (in Figure-5.5) with respect to $C(5, a)$ and input variable $i = 5$ then

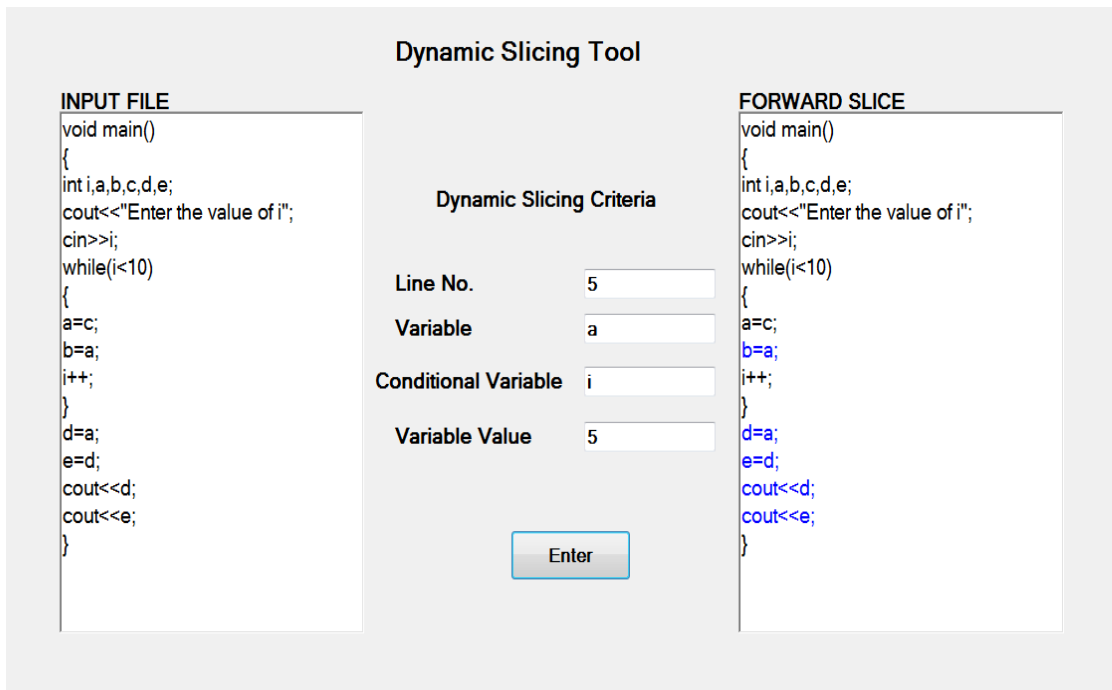


Figure-5.6

Dynamic forward slice of program (in Figure-5.5) with respect to $C(5, a)$ and input variable $i = 12$

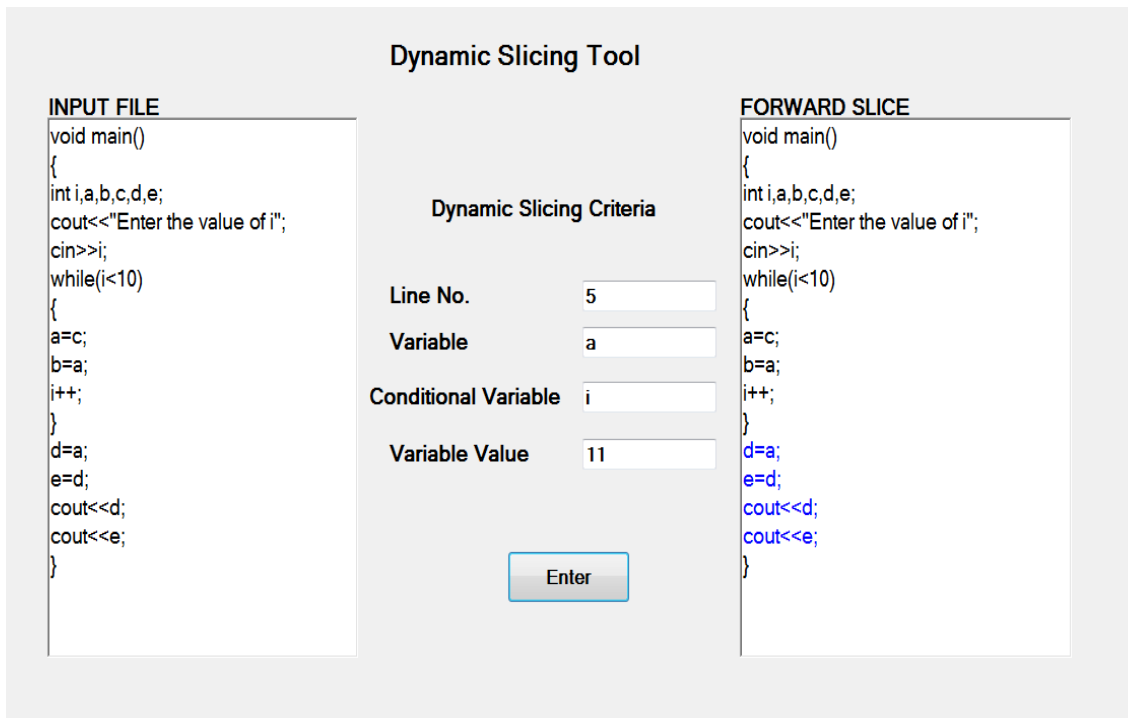


Figure-5.7

In these Figure find forward dynamic slice with respect to given slice criteria. And in these snapshot forward dynamic slice is highlighted in blue color with respect to slice

criteria. In this tool enter any program and line number in program and variable (for which find forward dynamic slice) and conditional variable and value of conditional variable in program as input then this tool give the forward dynamic slice with respect to given input as output.

Chapter-6
Conclusion and Future Work

6.1 Conclusion

In this thesis, dynamic forward slicing method was used to find program slices. A GUI tool has also been developed to demonstrate this method. Some of the features of the current approach are as follows

- This approach computes forward dynamic slice of a program without need of execution trace for tracking dependencies.
- This approach can minimize space complexity in comparison to backward approach of dynamic slicing.
- This approach computes forward dynamic slices of simple programs, which derives a slice for a particular variable affected by that variable.
- This approach can be used for analyzing and debugging the code.

6.2 Future Work

- This approach has not considered object-oriented features such as inheritance, polymorphism etc. so this approach can be extended for computing forward slices of object-oriented programs.
- This approach can extended for to find Forward Dynamic slicing of Concurrent Programs.

References

[1] Weiser M , " Program slicing" [J] IEEE Transaction on Software Eng, pp. 352-357 , 1994

[2] Weiser M , " Program slicing" Proceedings of the 5th International Conference on Software Eng, pp. 439-449 , 1981

[3] Zeller A. ," Tracking Dependencies " in Why Program Fails,2th edition : Elsevier Inc. , 2009

[4] Korel B. and Laski J., "Dynamic program slicing", Information Processing Letters, vol. 29, no. 3, pp. 155-163, 1988

[5] Korel B. and Laski J., "Dynamic slicing of computer programs", The Journal of Systems and Software, vol. 13, no. 3, pp. 187-195, 1990

[6] Venkatesh G. A., "The semantic approach to program slicing", ACM SIGPLAN Notices, vol. 26, no. 6, pp. 107-119, 1991

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

[8] Ferrante J. and Ottenstein K. and Warren J. , "The program dependence graph and its use in optimization" , ACM Trans Program Language System , vol. 9 , pp 319-349 ,1987

[9] Barpanda S. S., Biswal B. N., Ray M." Interprocedural slicing of generic programs ", International Conference on Signal Processing Systems , pp. 570-573 , 2009

[10] Song Y., and Huynh D., Forward Dynamic Object-Oriented Slicing, Application Specific Systems and Software Engineering and Technology(ASSET"99), IEEE CS Press, 1999

[11] Larsen L., Harrold M. J., Slicing Object-Oriented Software, Proceedings of the 18th international conference on Software Engineering, pp 495-505, 1996.

[12] Balagurusami E. "Object Oriented Programming with C++", 4th Edition, and Tata-McGraw Hill, 2007.

[13] Barpanda S. S., Biswal B. N., Ray M." Interprocedural slicing of generic programs ", International Conference on Signal Processing Systems , pp. 570-573 , 2009

[14] Weiser M., "Programmers use slices when debugging", Communication of ACM25, pp 446-452, July 1982.

[15] Ottenstein K. J., Ottenstein L. M., "The program dependence graph in a software development environment", Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pages 85-97, July 1995.

[16] Larsen L., Harrold M. J., "Slicing Object-Oriented Software", Proceedings of the 18th international conference on Software Engineering, pp 495-505, 1996.

[17] Agrawal H., Horgan J.. "Dynamic Program Slicing". Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, pp 246-256, 1990

[18] Zhao J., "Dynamic Slicing of Object-Oriented Programs", Tech-nical-Report SE-98-119, Information Processing Society of Japan, May 1998

[19] Song Y., Huynh D., "Forward Dynamic Object-Oriented Program Slicing", Application-Specific Systems and Software Engineering and Technology (ASSET '99), IEEE CS Press, pp 230-237, 1999

[20] Korel B., Yalamanchili S., "Forward Derivation of Dynamic Slices", Proceedings of the International Symposium on Software Testing and Analysis, pp. 66-79 , 1994

[21] Tibor G., " An Efficient Relevant Slicing Method for Debugging" , Software Engineering Notes, Software Engineering ESEC/FSE'99 Springer ACM SIGSFT, pp. 303-321, 1999

[22] Jianhong M., Min X., Hongtao W., Xiangling C., " A improved algorithm for forward computation of dynamic program slice ", World Congress on Software Engineering , pp. 374-378 , 2009

[23] Lucia A. De. ," Program slicing: methods and applications " , International Conference on Software Engineering, 2005

[24] Limin J. , Hongqiang J. , Jie L. ," A Dynamic program slice algorithm based on simplified dependence " , 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE) , vol(4) , pp. 356-359 , 2010

[25] Agrawal H., Richard A. De Millo , " Debugging with dynamic slicing and back tracking " software practice and experience , vol. 23 , no. 6 , pp. 589-616 , June 1993

[26] Lydia M. S. , Srinath I. and Gyani J. ,″ Static and Dynamic Attribute Slicing Tool for Object-Oriented Programs" , International Conference and Workshop on Emerging Trends in Technology , pp. 717-722, 2010

[27] Korel B. ," Computation of Dynamic Slices for Unstructured Programs." , IEEE Transactions on Software Engineering, pp. 17-34 , 1997

[28] Xuelian W. , Ruilian Z. , Lijian L. , "Dynamic slice algorithm for test data generation" , Computer Application , vol. 25 , No. 6 , pp. 1445-1450 , 2005.

[29] Barpanda S. S. , Biswal B. N. , Mohapatra D. , "A Graph coloring approach to slicing of object-oriented programs" , International Conference and Workshop on Emerging Trends in Technology , pp. 713-716 , 2010

[30] Horwitz S., Prins J., Reps T. ,"Dependence analysis of pointer variable", In Proceedings of the ACM 1989 Conference of Programming Language Design and Implementation, vol. 24, No. 7 ,1989

[31] Ottenstein K. , 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, vol. 19, No. 5, pp. 177-184, 1984

[32] Kamkar M. ,"Interprocedural Dynamic Slicing with application to debugging and testing", PhD Thesis, Linkoping University, 1993

List of Papers Presented

[1] Deepak Goyal, Mr. Karun Verma, Dr. Rajesh Bhatia "A Novel approach of forward dynamic slicing" , ACM SigSoft Software Engineering Notes , Sept Issue 2011 (Communicated)