DYNAMIC FORWARD SLICING

A thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology

in

Computer Science and Engineering

by

Amit Kumar Panda

(Roll no. 107cs014)

and

Praveen Kumar

(Roll no. 107cs028)

Under the guidance of :

Prof. D. P. Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela-769 008, Orissa, India

National Institute of Technology
Rourkela

Certificate

This is to certify that the project entitled, '**Forward Dynamic Slicing of Object Oriented Programs**' submitted by **Amit Kumar Panda** and **Praveen Kumar** is an authentic work carried out by them under my supervision and guidance for the partial fulfillment of the requirements for the award of **Bachelor of Technology Degree** in **Computer Science and Engineering** at **National Institute of Technology, Rourkela**.

To the best of my knowledge, the matter embodied in the project has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Date - 7/5/2011
Rourkela

(Prof. D. P. Mohapatra)
Dept. of Computer Science and Engineering

Abstract

Program slicing is a very important part of program development and maintenance . It is used for a number of applications such as program debugging, reverse engineering, software testing, software maintenance, etc. It is a programmers most important tool for debugging. We have come a long way since Weiser first introduced the concept of slicing. Initially, static slices were used but now mainly dynamic slices are being used. Programmers worldwide are busy trying to develop better and more efficient slicing techniques. In this paper we have proposed a new precise forward dynamic slicing algorithm.Our algorithm is based on marking and unmarking the stable and unstable edges in the PDG according to their execution. We have calculated slices by using intermediate representation in the form of program dependency graph(PDG). We have intoduced modified notions of stable and unstable edges and used them to draw MPDG(Modified Program Dependency Graph). We have tested our algorithm by taking two sample programs. Our research has been confined to simple C and C++ programs.

Acknowledgments

We express our profound gratitude and indebtedness to **Prof. D. P. Mohapatra**, Department of Computer Science and Engineering, NIT, Rourkela for introducing the present topic and for their inspiring intellectual guidance, constructive criticism and valuable suggestion throughout the project work. We truely value his esteemed motivation and guidance from beginning to end of the thesis. It would not have been possible on our part to complete the thesis without his priceless advises and assistance.

Date - 7/5/2011
Rourkela

Amit Kumar Panda

Praveen Kumar

Contents

List of Figures

Chapter 1

Introduction

Program slicing means reducing the given program to a minimal number of statements with respect to a given criteria which is the variable and the number of statement in the program. Program slicing is used for a large number of computer applications such as debugging, maintenance, testing, etc. For calculating slices various graph visualizations are used which are called intermediate representation such as CFG(control flow graph) , PDG(programd dependency graph) etc. Slicing is of various types such as static slicing, dynamic slicing, forward slicing, backward slicing. A large numbler of algorithms for calculating slices have been proposed . However , this is a recently opened stream so all the work has been in its early stages . So , the present algorithms have a lot of drawbacks such as they dont cover all types of programs, consume a lot of space and time, can get into non-responsive state, etc. The efficiency of a algorithm can be computed in terms of space and time complexity and the range of program it covers. Also, the accuracy of the algorithm is very much important. [1]

1.1 Motivation

Program slices are used for a number of computer applications such as program testing, debugging, etc. These slices are indispensable for development of programs. Hence, programmers all over the world are striving to develop efficient, better and speedy slicing techniques.

1.2 Objective

Keeping above mentioned objective in mind, we are trying to develop a new algorithm to find precise forward dynamic slices in an efficient and faster way. We do so by developing an intermediate representation in the form of Program Dependence Graph and its modified forms.

Chapter 2

Basic Concepts

We explain some basic terms and definations related to slicing in the following sections.

2.1 Slices and types of Slices

The input elements for a slice are the line of code and the variable w.r.t which the slice is to be computed. It is represented as <S,V>. A slice is the minimal program that is effected by the given criterion.[1]
Types of Slices:

(1)Static Slice: The slice which is computed for a general set of variables are called static slices i.e, static slices are the slices for the whole range of values of the variables involved in the program.[1]

(2)Dynamic Slice: The slice which is computed for a given set of values are called dynamic slices i.e, these are very specific. Thus these are very short compared to static slices. Nowadays, dynamic slices are used because these are easier to construct , faster to execute and pinpoint the errors in the program.[1]

(3)Precise Dynamic Slice: A precise dynamic slice is a dynamic slice, which contains the least amount of statements possible, for the given criterion[9].

(4)Forward Dynamic Slice : When we are given a particular slicing criterion then the slice which shows which statements and variables will be affected by the given criterion is known as Forward Dynamic Slice.[1]

(5)Backward Dynamic Slice: When we are given a particular slicing criterion then slice which shows which statements and variables have affected the given criterion is known as Backward Dynamic Slices.[1]

2.2 Dependency

Each statement of a code is dependent on other statement in some way , this is known as dependency.Basically, it is of two types:
(1)Data Dependency: When a statement or a variable is dependent on some other statement for some data it is known as data dependency.[1][2][10]

(2)Control Dependency: When the execution of a statement is dependent on some other statement it is called as control dependency.[10]

2.3 Visualisation of Slices

Visualisation of slices is a very efficient technique of understanding and developing slices. It is done in following ways:
(1)Control Flow Graph : It is simple representation of control flows and thus the flow in which statements are executed.

(2)Program Dependence Graph: It is representation of the various dependencies among the statements of a program. Wediscuss PDG with following example program:

1.int a,b;

```
2.b=5;
3.if(a>b)
4.a = a + 1;
5.while(a>b) do
6.a = a - 1;
else
7.write(a);
```

PDG for the program is as follows:

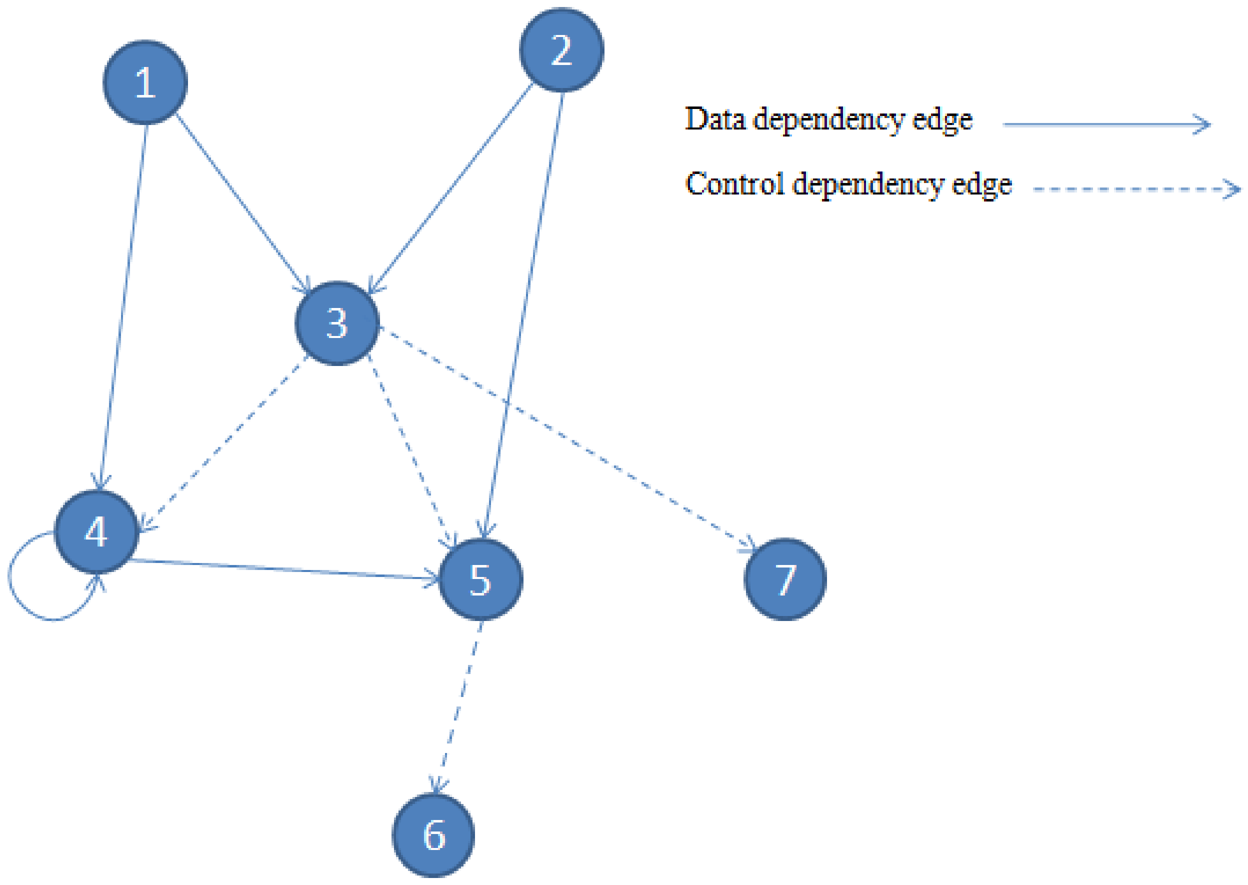

Figure 2.1: PDG of Example

Chapter 3

Review of related work

Weiser[12] first introduced the idea of slicing. He introduced the idea of static slicing.He used control flow graph to compute slices. The major disadvantage of his approach was that each slice was computed from beginning i.e, during computation of slices nothing was saved or stored for future use. Then Ottenstein and Ottenstein[15] introduced the idea of PDG(program dependency graph) and used it to compute intraprocedural slices.This was a major breakthrough in the field of slicing. Horwitz[4] took this idea further to SDG (System Dependency Graph) and computed interprocedural slices. Then Korel and Laski[14] introduced the concept of dynamic slices. This was another important leap for slicing. They however used Weiser CFG for computing slices. The method used by Korel and Laski becomes useless when there are loops in the program.For the first time Agrawal and Horgan[16] used dependence graphs to compute dynamic slices. They also introduced the idea of precise dynamic slices and proposed DDG(Dynamic Dependency Graph) for computing precise dynamic slices. In this a new node is created for each executed node and its associated nodes .Mund[9][10] et. proposed the concept of stable and unstable edges and use them to create dynamic slices. They further improved their algorithm and proposed a edge marking unmarking algorithm and also node marking and unmarking algorithm. They proved that their algorithms are better than others in terms of precision , time complexity and space complexity. Most of these algorithms calculate backward

slices. Much of the literature on program slicing is concerned with improving the algorithms of slicing keeping in mind reducion of the size of the slice and improvment the efficiency of computation. All the works focus on computation of precise dependence information and the accuracy of the computed slices. The approach of Weiser[12] for intraprocedural static slicing worked on iteratively solving data-flow equations representing influences between statements. Weiser[12] used the control flow graph (CFG) as the intermediate representation for his static slicing algorithm. Later Weiser presented an algorithm, which has two phases for computing inter-procedural slices. Ottenstein and Ottenstein presented a linear time solution for intraprocedural static slicing focusing on graph reachability in the program dependence graph (PDG). Horwitz et al. extended the representation by PDG to system dependence graph (SDG) for inter-procedural static slicing. Hwang et al. presented an inter-procedural static slicing algorithm which is based on replacing the recursive calls by instances of the body of the procedure.

Korel and Laski[14] extended Weisers static slicing algorithm to the dynamic slicing cases. They computed dynamic slices by using data-flow equations. This method needs O(N) space to store the history of the executions, and O(N squared) space to store the dynamic flows of data, where N is the number of statements. Note that N is unbound for program containing loops.

Agrawal and Horgan[16] were the first to present algorithms for finding dynamic program slices using PDG. They first used PDG as the intermediate representation and marked the nodes of this graph as the corresponding parts of the program are executed for a given input set. The algorithm of Ottenstein and Ottenstein[15] for static slicing is applied to the subgraphs of the PDG to compute the dynamic slices induced by the marked nodes. This approach is very much imprecise because it does not consider the situations where there exists an edge in the PDG from a marked node u to a marked node v but the definition at v is not used at u.We show this kind of imprecision through an example. Consider the following program:

.

```
Integer m,a,I,b,x,y,z;
1. read(m);
2. a=0;
3. i=1;
4. b=2;
5. while(i<=m) do
6. read(x);
7. if (x<=0) then
8. y=x +5;
else
9. y= x-5;
10. z= y+4;
11. if (z>0) then
12. a= a+z;
else
13 b=a+5;
14. i=i+1;
Endwhile
15. write(a);
16. write(b);
```

Let us draw the PDG of above program.

.

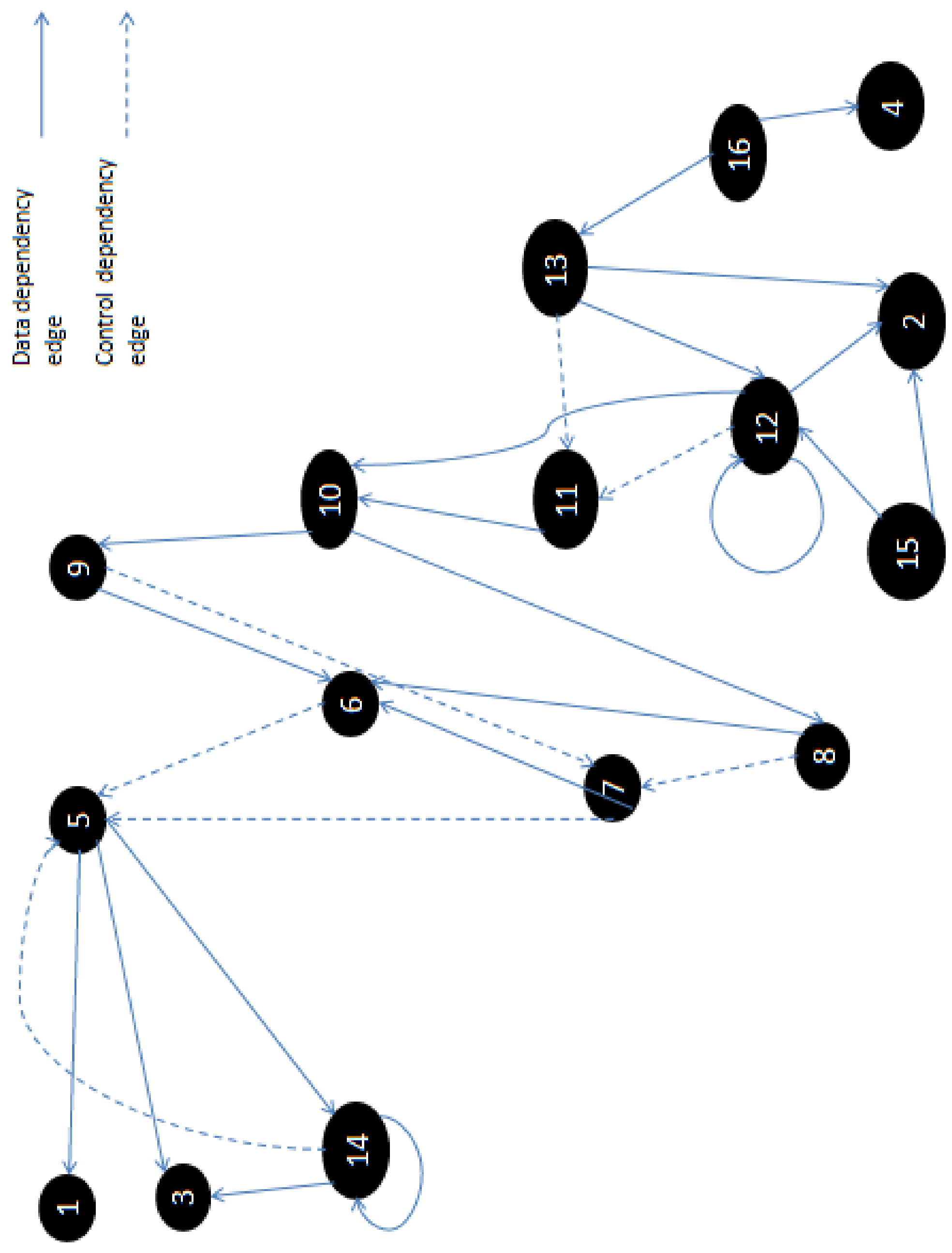

Figure 3.1: PDG of the Example

.

Let the input m = 2, and x in the first and second iterations be 0 and 2, respectively. In first iteration of the while loop, the statement 8 defines a value for y. In second iteration of the loop, the statement 9 here defines a value of y without using its previous value, and the previous definition of y is destroyed. Therefore, the dynamic slices for the slicing criterion<10,z> in the second iteration of the while loop here it should contain the statement 9 and it should not contain the statement 8. Let us find the dynamic slice using first approach of Agrawal and Horgan[16]. We mark the node 8 in first iteration of the loop and node 9 in the second. As the node 10 has two outgoing dependence edges to the nodes 8 and 9 in the PDG, the statements 8 and 9 get included in the slice, which is very much imprecise. The second approach of Agrawal and Horgan[16] marks the edges of the PDG as and when the corresponding dependence arise during program execution. The dynamic slice is computed by applying the static slicing algorithm of Ottenstein and Ottenstein[15] and traversing the PDG only along all the edges which are marked. This approach finds accurate dynamic slices of programs having no loops. Whenever the loops are present, the slices may include more statements than those which are actually necessary, because this approach does not consider the fact that execution of the same statements at different iterations of a loop may be (transitively) dependent on different sets of statements. Agrawal and Horgan[16] pointed out that their second approach for computing dynamic slices produces results identical to that produced by the algorithm of Korel and Laski. Note that the PDG of a program having n number of statements requires only O(n squared) space. So, the space requirement of Agrawal and Horgans second algorithm is O(n squared). But the algorithm of Korel and Laski may use unbounded space in worst case.
The disadvantages of the second approach by Agrawal and Horgan motivated their third approach: construct a dynamic dependence graph(DDG) creating a new node for each occurrence of a statement in the execution

history along with the associated dependence edges. The negativeness of using the DDG is that the total number of nodes equals to the number of statements executed which may not be bounded for programs having loops.

In their fourth approach ,Agrawal and Horgan[16] proposed to reduce the number of nodes present in the DDG by merging the nodes whose transitive dependences map to the same set of statements. Alternatively, a new node is introduced only if it can create a new dynamic slice. This check adds to run-time overhead. This reduced graph is called the reduced dynamic dependence graph(RDDG). The size of this RDDG is proportional to the number of dynamic slices that may arise.. The number of slices of the program is O(2 raise to n) in the worst case(where n is the number of statements).

Chapter 4

Slicing Algorithms

Here we propose a precise forward dynamic slicing algorithm: Intermediate Representation Used: Program Dependency graph(PDG) and Modified Program Dependency Graph(MPDG)

4.1 Terms and Definations Used

(1)Unstable edge:
(a)All conditional control dependency edges are unstable.
(b)If S is some statement of a program P then an outgoing dependency edge(Si,S), in the PDG of P is said to be unstable if there exists an outgoing dependency edge(Sm,S) or a self loop (S,S) with Si not equal to Sm such that Sl and Sm both define same variable.[9]

(2)Stable edge:All other edges are stable edges.[9]

4.2 Proposed Algorithm

Step 1: Construct PDG (Program Dependency Graph) of the program.
Step 2: Construct MPDG(Modified Program Depenedency Graph) of the program which contains only stable and unstable edges according to their definitions.

Step 3:Mark all the stable edges and unmark all the unstable edges before running the program. Also unmark all the unstable edges before each iteration of a conditional loop to compute the slices for that iteration.
Step 4: Now execute the program . Mark the unstable edges according to their most recently used definition.
Step 5:Now if there are two or more nodes forming a cycle of dependency combine those nodes and form a single node. Now, all incoming edges to the individual nodes are directed towards this new combined node and all outgoing edges from each node will be shown as outgoing edges from new node.Do this for all the cyclic dependencies arising. Also remove all the self loops. Do this after each time program is executed.
Step 6: Compute slices for the desired node using algo compSlice(node n)

```
compSlice(node n)
{
Set dslice=NULL
If node n is not traversed
Mark node n as traversed
For each outgoing dependency edge
Add the node m to dslice
And for each such node m do compSlice(node m)
}
```

Example program 1:

```
integer a,b,c;
1.read(a)
2.b = 1
3.c = 4
4.while(b <= a) do
5. if((b mod 2) > 0) then
6. c = c + 9
```

```
else
7. c = 10
8.write(c)
9.b = b + 1
endwhile
```

figure:

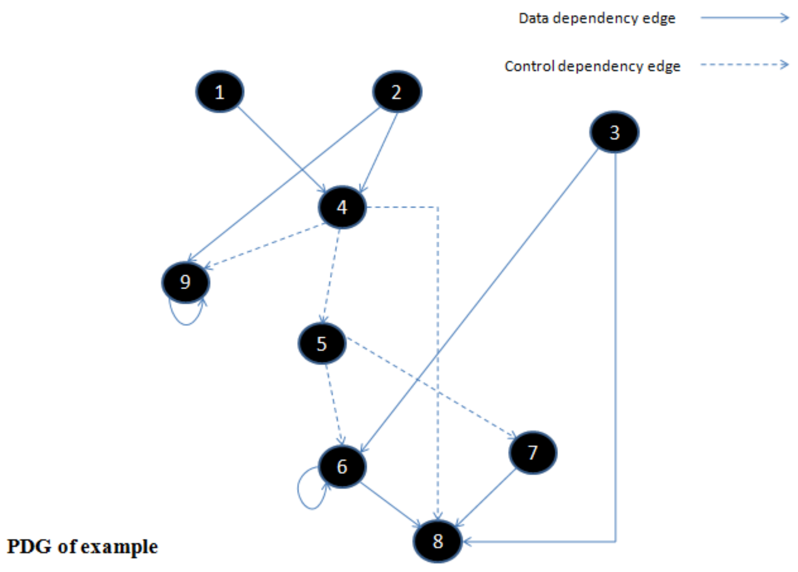

Figure 4.1: PDG of Example 1

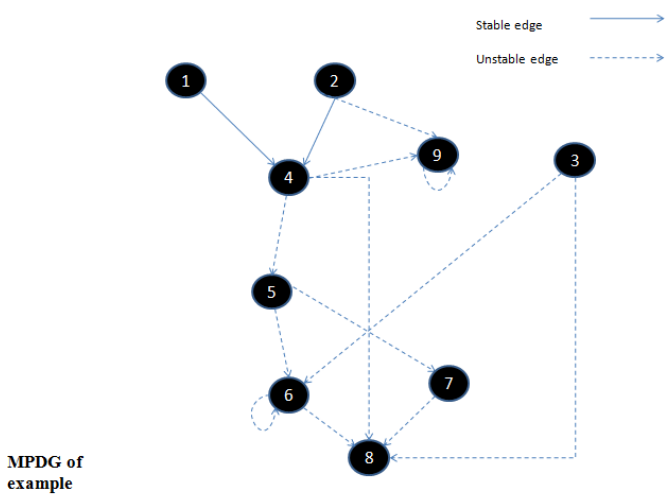

Figure 4.2: MPDG of Example 1

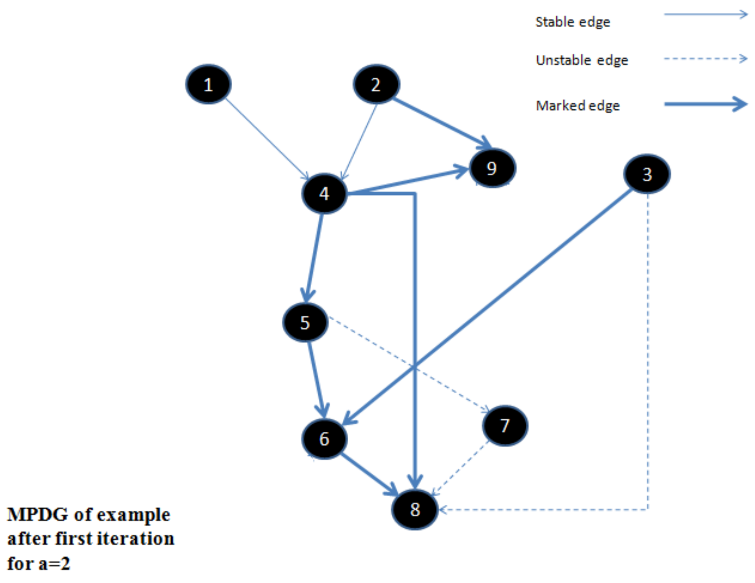

Figure 4.3: MPDG of Example 1 after 1st iteration

First Iteration: <2,b>= 2,4,5,6,8,9

i.e, slices for 2. b=1 ::

integer a,b,c;

1.read(a)

<u>2.b = 1</u>

3.c = 4

```
4.while(b <= a) do
5. if((b mod 2) > 0) then
6. c = c + 9
else
7. c = 10
8.write(c)
9.b = b + 1
endwhile
```

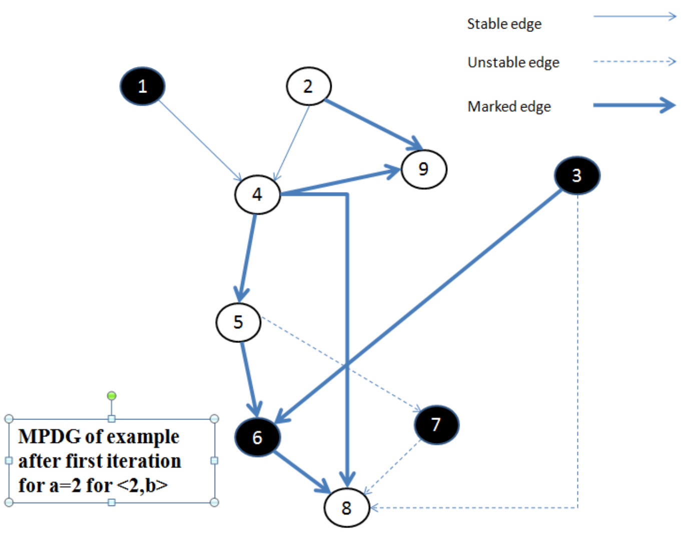

Figure 4.4: MPDG of Example 1 after 1st iteration for <2,b>

<5,b>=5,6,8

i.e, slices for 5. if(b mod 2)>0) then ::

5. if(b mod 2)>0) then

6. c=c+9

8. write(c)

Now we look forward to the second iteration.

Second Iteration:

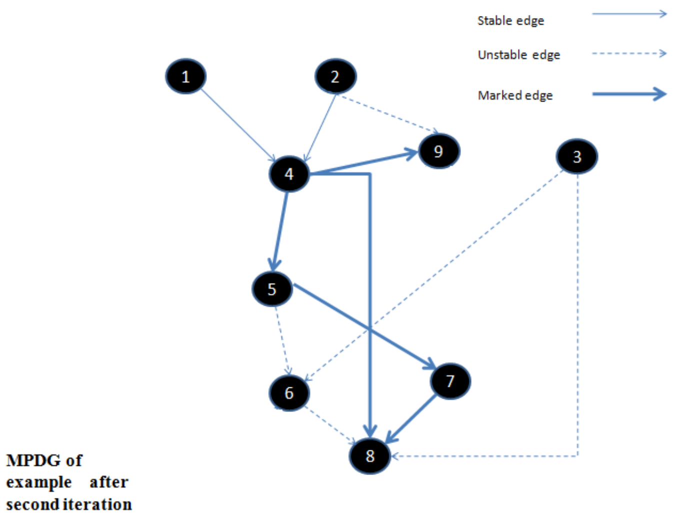

Figure 4.5: MPDG of Example 1 after 2nd iteration

<3,c> = 3

<4,b>=4,5,7,8,9

i.e, slices for 4. while(b<=a) do::

4. while(b<=a) do

5. if(b mod 2)>0) then

7. c=10

8. write(c)

9.b=b+1

Again slices for <2,b>=2,4,5,8,7,9.

i.e.

integer a,b,c;

1.read(a)

<u>**2.b = 1**</u>

3.c = 4

<u>**4.while(b <= a) do**</u>

<u>**5. if((b mod 2) > 0) then**</u>

6. c = c + 9

else

<u>**7. c = 10**</u>

```
8.write(c)
9.b = b + 1
endwhile
```

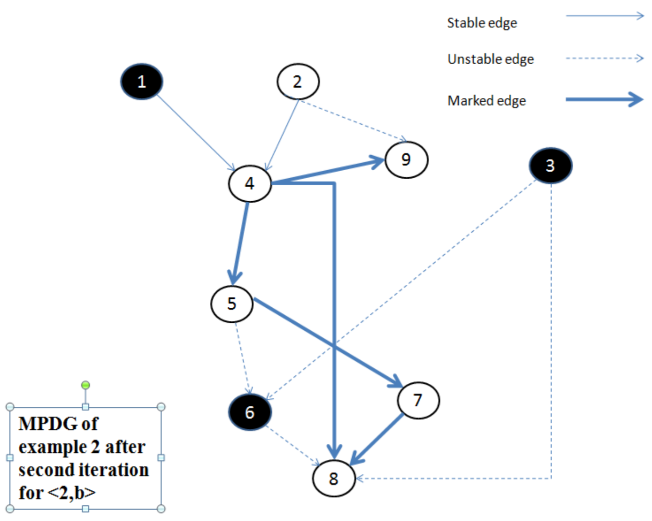

Figure 4.6: MPDG of Example 1 after 2nd iteration for <2,b>

Example program 2:

```
integer m,a,i,b,x,y,z;

1.read(m);
2.a = 0;
3.i = 1;
4.b = 2;
5.while(i <= m) do
6.read(x);
7.if(x <= 0) then
8.y = x + 5;
else
9.y= x - 5;
10.z= y + 4;
11.if(z>0) then
12.a= a + z;
else
13.b= a + 5;
14.i= i + 1;
endwhile
15.write(a);
16.write(b);
```

Precise Dynamic forward Slices for m=2

Now lets analyse the calculation of dynamic slices step by step. Lets choose the slicing criteria as <7,x>. Now for first iteration x=-6 , hence, statement 7 will be executed and statement 8 will not be executed. Therefore, the unstable edge(7,8) will be marked and the unstable edge (7,9) will not be marked as shown in the MPDG after first iteration. Further, statement 10

uses the value of y defined at statement 8. Hence, the edge (8,10) will be marked and edge(9,10) will not be marked. Now the value of z becomes >0 therefore statement 12 will be executed and statement 13 will not be executed and so they will be marked and unmarked respectively. The first iteration completes after statement 14. Now looking at the MPDG after first iteration and applying our compSlice(node n) algorithem we can easily find out the slices. We start at the node 7 , it has one outgoing edge(7,8) (remember we have to take into account only stable edges and marked unstable edges) .Thus, statement 8 is added to the dslice. Now , start at 8 , it has also only one outgoing edge(8,10) thus 10 is added to dslice. Similarly, (10,11) , (11,12) are traversed and added to dslice.Thus , finally we get dslice<7,x> = 7,8,10,11,12 For second iteration we have x=7 so statement 9 will be executed and statement 8 will not be executed and so they will be marked and remain unmarked respectively. Now statement 10 will use value of y defined at 9 so edge(9,10) will be marked. Similarly, (11,12) will be marked. Now , statement 15 will also be executed and it will use value of a defined at 12 hence edge(12,15) will be marked. Thus, we get final dslices<7,x> = 7,9,10,11,12,15

After first iteration for x=-6
<7,x> = 7,8,10,11,12,14

```
integer m,a,i,b,x,y,z;

1.read(m);
2.a = 0;
3.i = 1;
4.b = 2;
5.while(i <= m) do
6.read(x);
7.if(x <= 0) then
8.y = x + 5;
else
9.y= x - 5;
```

```
10.z= y + 4;
11.if(z>0) then
12.a= a + z;
else
13.b= a + 5;
14.i= i + 1;
endwhile
15.write(a);
16.write(b);
```

<5,m> = 5,6,7,8,10,11,12,14

i.e, slices for 5.while(i <= m) do(in figure 4.9)

```
5.while(i <= m) do
6.read(x);
7.if(x <= 0) then
8.y = x + 10.z= y + 4;
11.if(z>0) then
12.a= a + z;
14.i= i + 1;
```

similarly <11,z>=11,12

After second iteration for x=7

<7,x> = 7,9,10,11,12,15

i.e, slices for 7.if(x <= 0) then

```
    integer m,a,i,b,x,y,z;

1.read(m);
2.a = 0;
```

```
3.i = 1;
4.b = 2;
5.while(i <= m) do
6.read(x);
7.if(x <= 0) then
8.y = x + 5;
else
9.y= x - 5;
10.z= y + 4;
11.if(z>0) then
12.a= a + z;
else
13.b= a + 5;
14.i= i + 1;
endwhile
15.write(a);
16.write(b);
```

similarly, <5,m> = 5,6,7,9,10,12,15

<11,z>= 11,12, 15

.

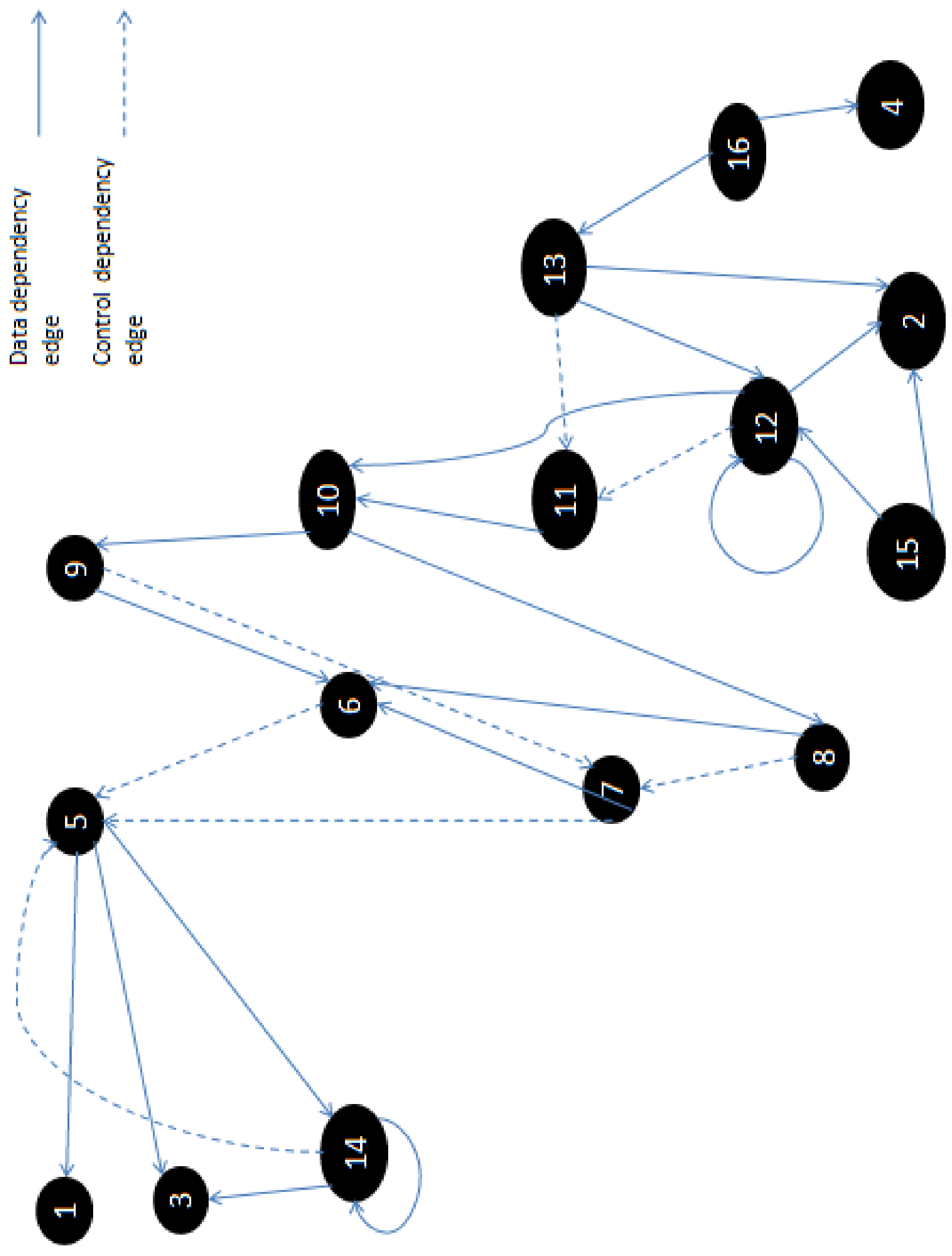

Figure 4.7: PDG of Example 2

.

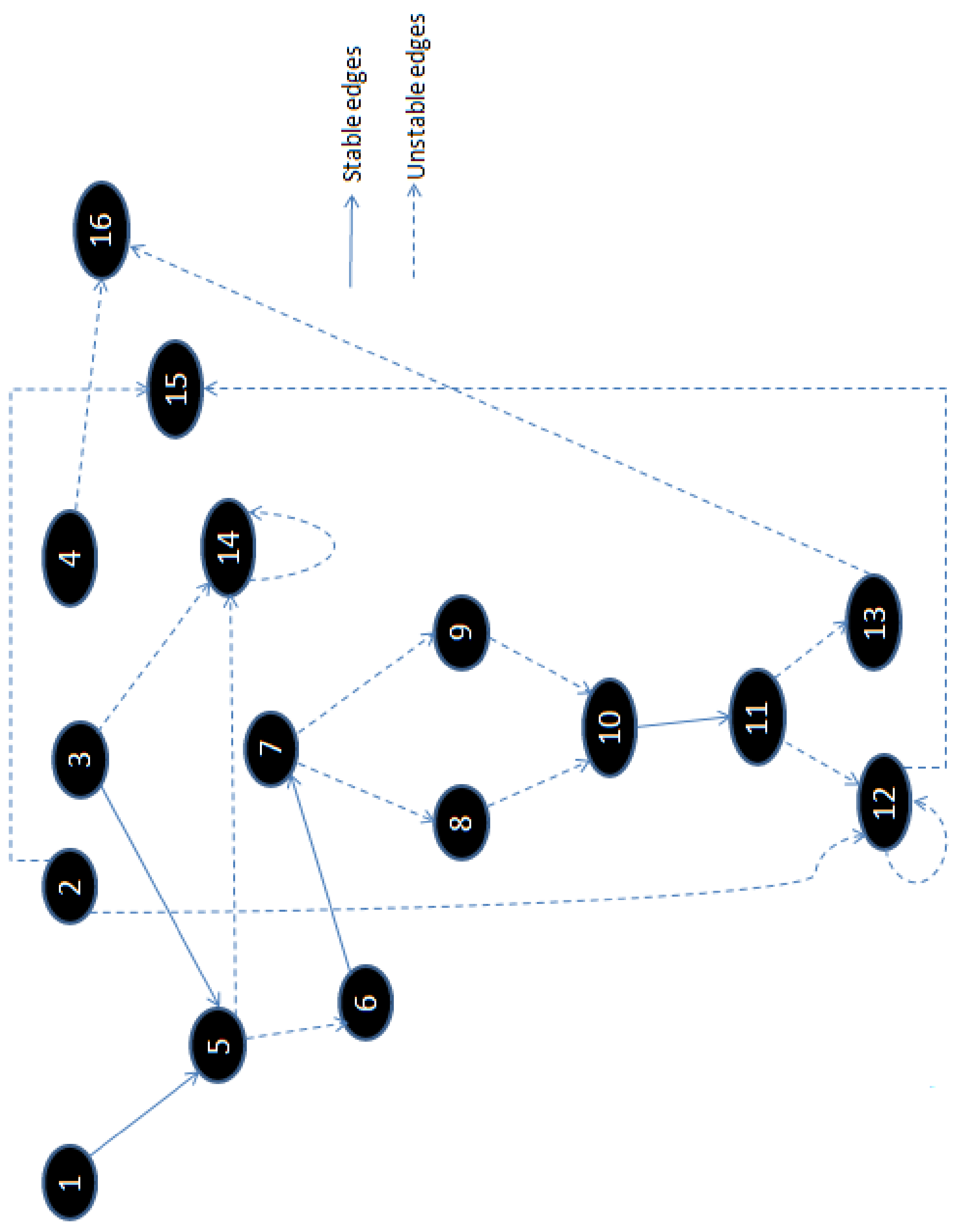

Figure 4.8: MPDG of Example 2

.

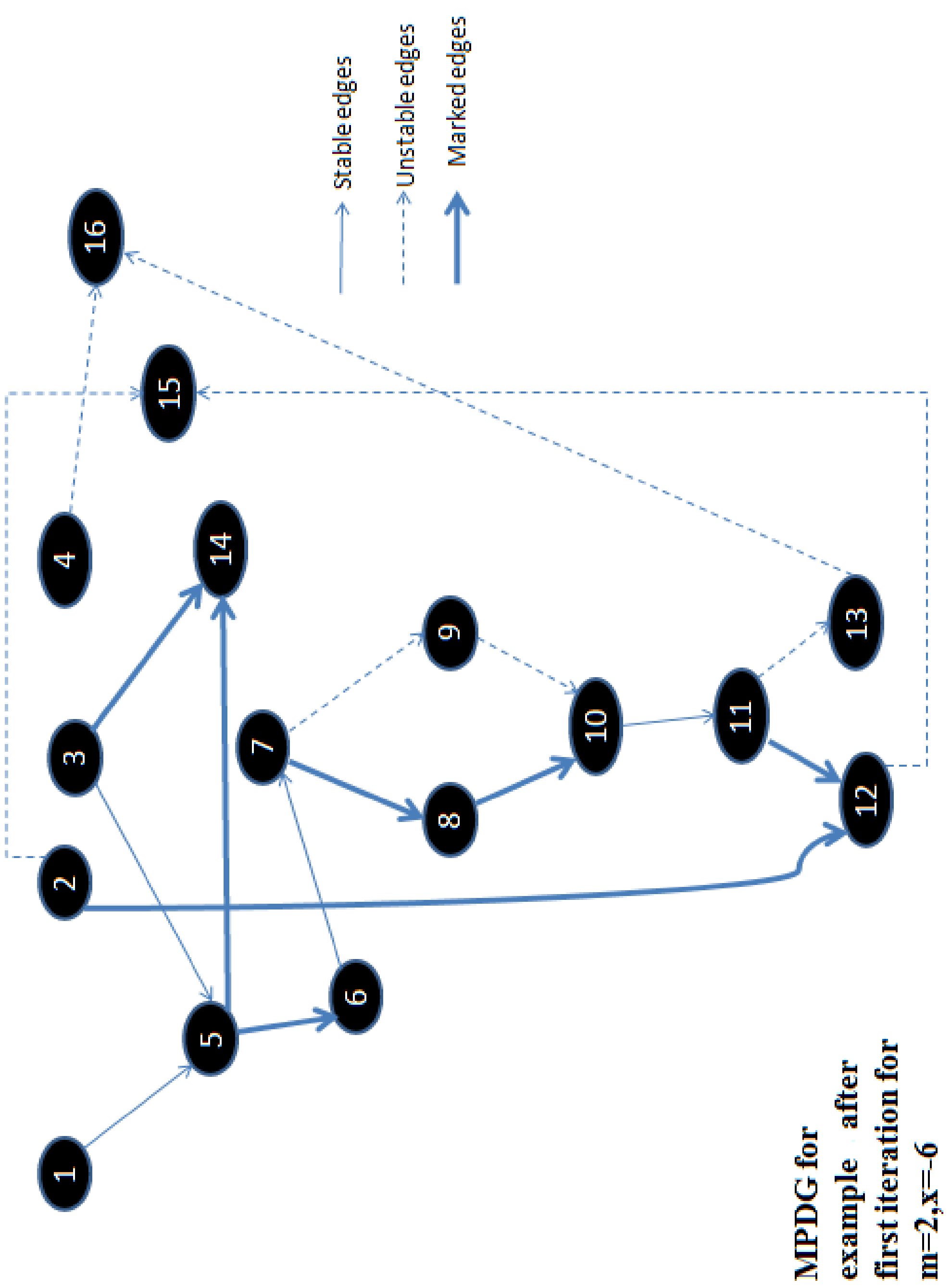

Figure 4.9: MPDG of Example 2 after 1st iteration

.

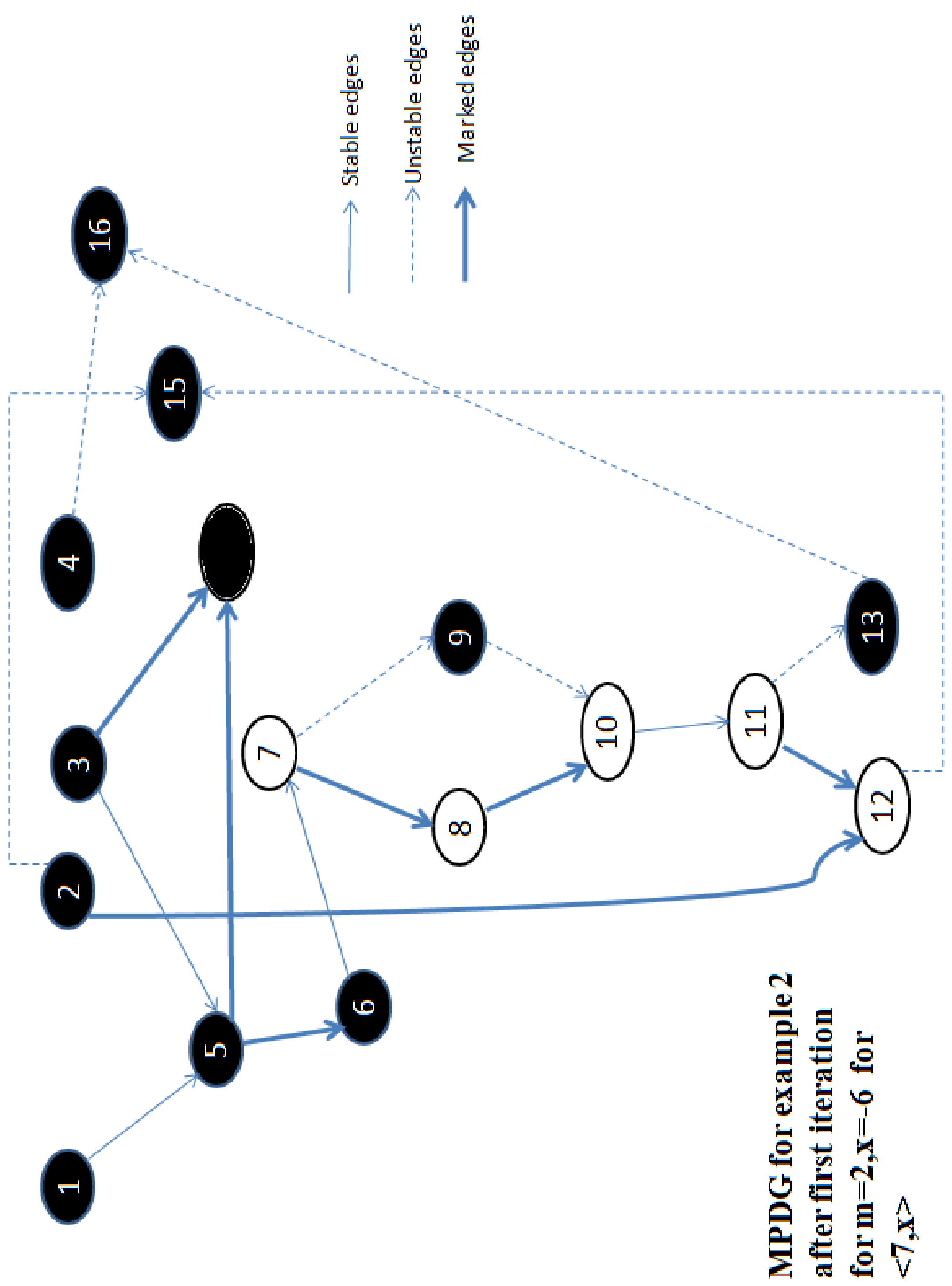
1
2
3
4
5
6
7
8
9
10
11
12
13
15
16
Stable edges
Unstable edges
Marked edges
MPDG for example 2 after first iteration for m=2,x=-6 for <7,x>

Figure 4.10: MPDG of Example 2 after 1st iteration for <7,x>

.

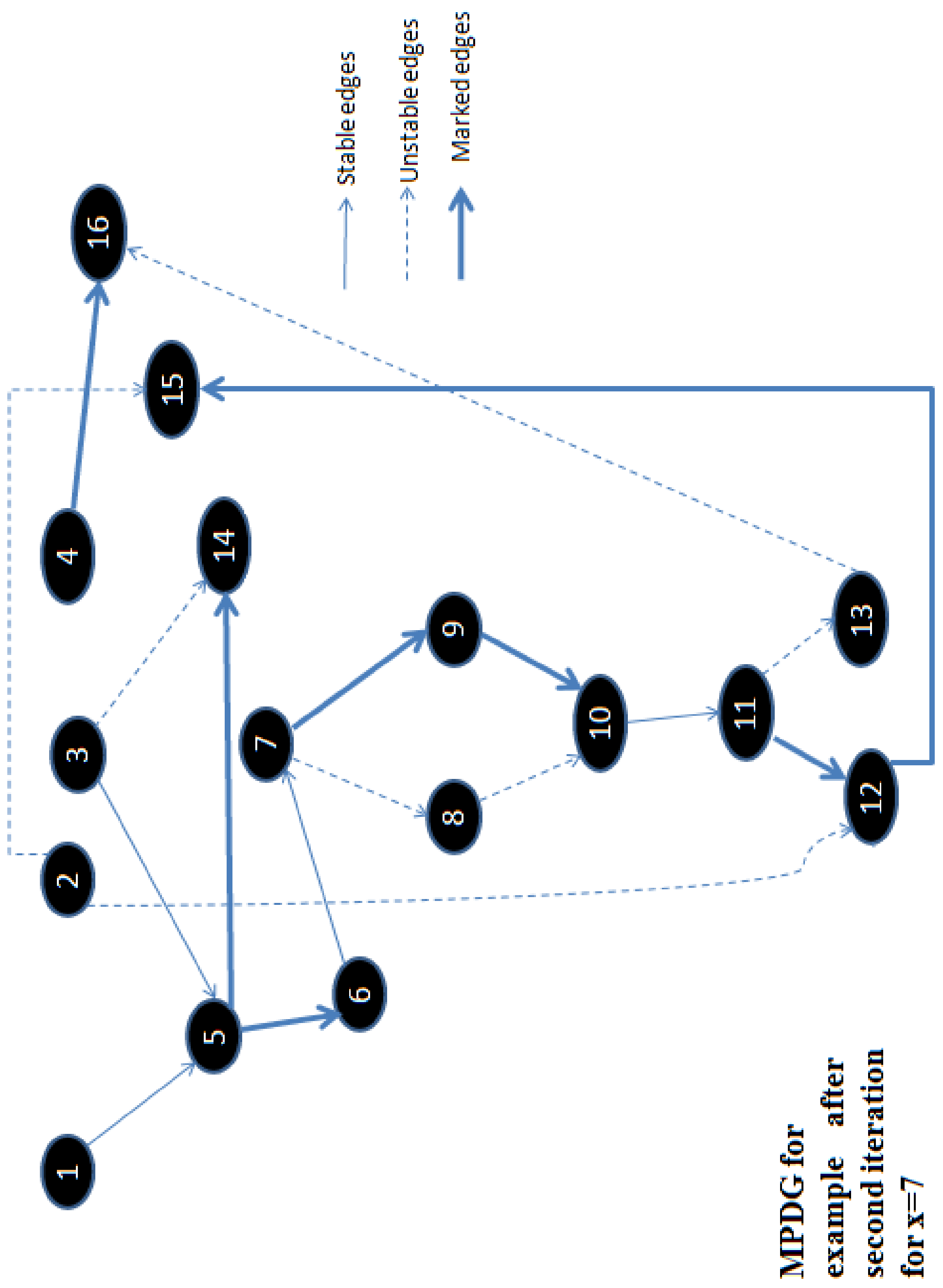

Figure 4.11: MPDG of Example 2 after 2nd iteration

.

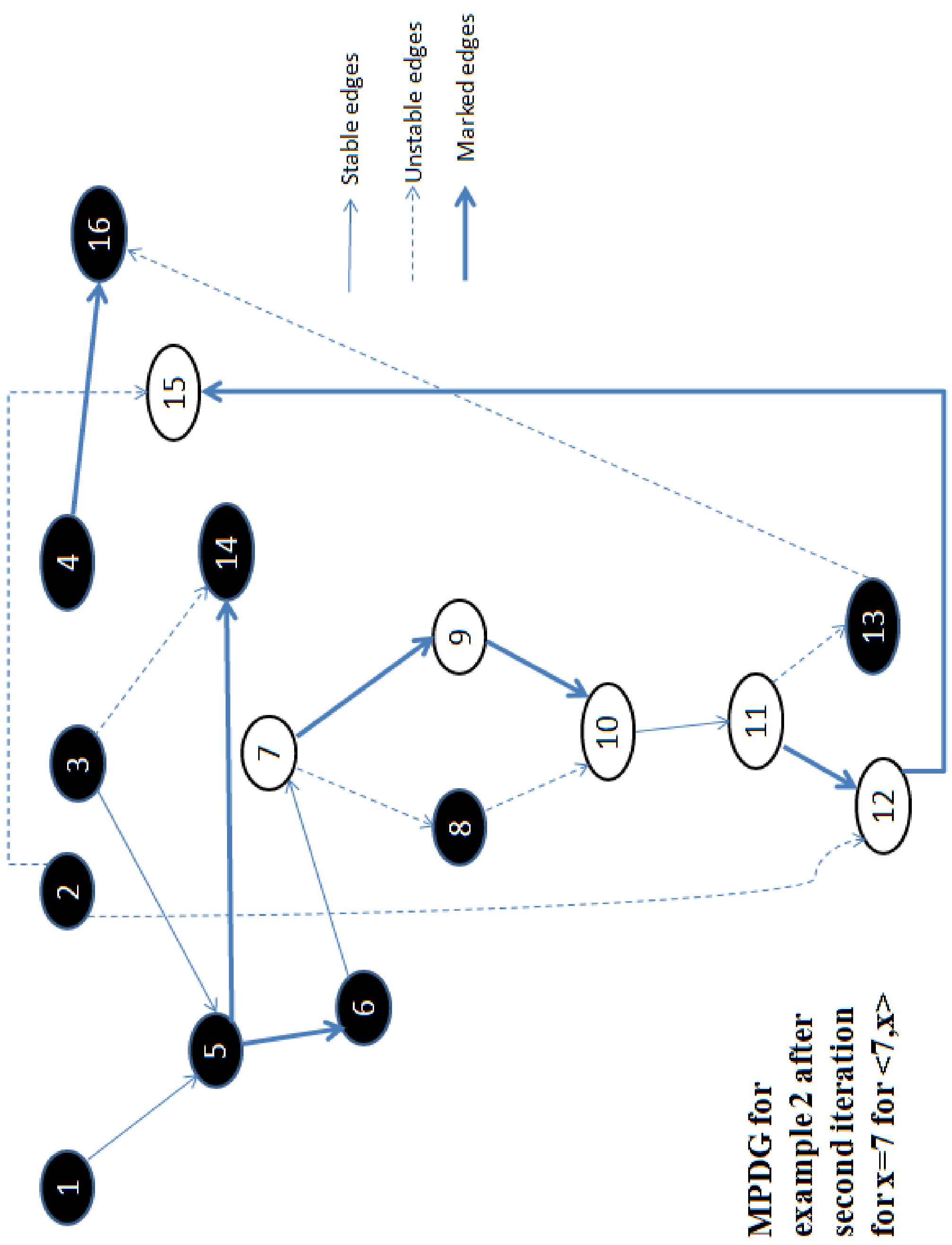

Figure 4.12: MPDG of Example 2 after 2nd iteration for <7,x>

Chapter 5

Conclusion and Future Work

5.1 Conclusion

We defined stable and unstable edges for precise forward dynamic slices. We empirically verified our precise forward dynamic slicing algorithm on two sample programs and presented a few prcised dynamic slices. We used intermediate representation in the form of PDG(Program Dependency Graph) and its modified form(MPDG).

5.1.1 Comparision with Other Algorithms

1.Most of the present algorithms such as algorithms proposed by Agrawal and Horgan[16]"Don't compute precise dynamic slices", but we have computed precise forward dynamic slices.
2.Most of the algorithms such as algorithm proposed by Ottenstein and Ottenstein reaches a non-responsive state when there are loops in the PDG we have overcome that drawback also.

5.2 Future Work

Since there are large number of unstable edges and we have to draw modified PDG, the algorithm consume a lot of space and time. Thus , there is a lot of scope for further development w.r.t space and time complexity.Our

proposed algorithm can be extended further to work on object oriented features in Java and C++.

Chapter 6

References

1.Loren Larcen and Mary Jean Harrold , "Slicing Object Oriented Software", page 495 - 505 (1996).

2.David W Binkley and Keith Brian Gallagher, "Program Slicing", Advances in Computers Volume 43, page 1-45, (1996).

3.H.Agrawal Slicing programs with jump statements.In Preceedings of SIGPLAN94 Conference on programming language design and implementation, pages 60-73 (1994).

4.S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages , pages 384-396(1993).

5.J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems , volume-9,Issue-3,pages 319-349,(1987).

6.K. B. Gallagher and J. R. Lyle Using program slicing in software maintenance IEEE Transactions onSoftware Engineering, volume-17,Issue-8,pages 751-761(1991).

7.Hemant D. Pande and Barbara G. Ryder Static type determination for C++,Proceedings of the 6th conference on USENIX Sixth C++ Technical Conference - volume 6,page 5-5(1994).

8.Jian-jun Zhao Dynamic Slicing of Object Oriented ProgrammingWuhan University Journal Of Natural Sciences volume-6,Issues 1-2,pages 391-397(2001).

9.G. B. Mund, R. Mall, S. Sarkar ”An efficient dynamic program slicing technique”,Department of Computer Science and Engineering,IIT Kharagpur,Information and Software Technology(44), pages 123-132(2002).

10.G. B. Mund, R. Mall, S. Sarkar ”Computation of intraprocedural dynamic program slices”,Department of Computer Science and Engineering,IIT Kharagpur,Information and Software Technology(45), pages 123-132(2003).

11.J. A. Dallal ”An Efficient Algorithm for Computing all Program Forward Static Slices”, World Academy of Science, Engineering and Technology(16)(2006).

12.M. Weiser, ”Programmers use slices when debugging”,Communication of the ACM 25(7),pages 446-452(1982). 13.M. Weiser,”Program Slicing” IEEE Transactions on Software Engineering 10(4), pages 352-357 (1984).

14.B. Korel, S. Laski ”Dynamic Program Slicing”, Information Processing letters,29(3),pages 155-163(1988).

15.K Ottenstein and L. Ottenstein ”the Program Dependence Graph in Software Development Environment”,Procedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symphosium on Practical Software Development environments, SIGPLAN Notices, pages 177-184(1984).

16.H. Agrawal and J. Horgan”Dynamic Program Slicing”,Proceedings of ACM SIGPLAN 90 conference on Programming Language Design and Implementation, SIGPLAN Notices analysis and verification(6),pages 246-256(1990).