International Journal of Rough Sets and Data Analysis
Volume 6 • Issue 1 • January-March 2019

Conditioned Slicing of Interprocedural Programs

Madhusmita Sahu, National Institute of Technology, Odisha, India

ABSTRACT

Program slicing is a technique to decompose programs depending on control flow and data flow amongst several lines of code in a program. Conditioned slicing is a generalization of static slicing and dynamic slicing. A variable, the desired program point, and a condition of interest form a slicing criterion for conditioned slicing. This paper proposes an approach to calculate conditioned slices for programs containing multiple procedures. The approach is termed Node-Marking Conditioned Slicing (NMCS) algorithm. In this approach, first and foremost step is to build an intermediate symbolization of a given program code and the next step is to develop an algorithm for finding out conditioned slices. The dependence graph, termed System Dependence Graph (SDG), is used to symbolize intermediate presentation. After constructing SDG, the NMCS algorithm chooses nodes that satisfy a given condition by the process of marking and unmarking. The algorithm also finds out conditioned slices for every variable at every statement during the process. NMCS algorithm employs a stack to save call context of a method. Few edges in SDG are labeled to identify the statement that calls a method. The proposed algorithm is implemented, and its performance is tested with several case study projects.

KEYWORDS

Conditioned Slice, Dynamic Slice, Program Slicing, Static Slice, System Dependence Graph

Note: This is an extended version of the paper published in *Proceedings of 3rd International Conference on Computational Intelligence in Data Mining (ICCIDM 2016)*, Bhubaneswar, 2016 (Sahu et al., 2016).

1. INTRODUCTION

Program slicing is a decomposition technique utilized to decompose programs depending on control flow and data flow amongst several lines of code in a program code. It is a kind of program analysis technique. It takes out statements related to computation of a variable's value at a specified point in program. The pulled out statements, containing assignment and predicate statements, constitute a program slice. These statements may affect or be affected by value of variable v at program location l. A slice is computed by employing a slicing criterion. The tuple $< l, v >$ is regarded as a slicing criterion. The slice may be static or dynamic according to input to program code. It is said to be static when it extracts all statements from a program code w.r.t. a slicing criterion regardless input to program (Weiser, 1981). On the other hand, it is said to be dynamic when all statements from a program are extracted w.r.t. a slicing criterion for a specific input to program code (Korel & Laski, 1988).

Program slices are computed in two steps. The first and foremost step is concerned with the construction of an intermediate symbolization of program code. In next step, an algorithm is applied

DOI: 10.4018/IJRSDA.2019010103

to that intermediate representation to find out slices. Program slicing has been employed in many areas of software engineering like debugging, software maintenance, testing, functional cohesion, software refactoring, software quality assurance, etc.

Conditioned slicing is a generalization of static slicing and dynamic slicing (Canfora et al., 1998). A conditioned slice is in the form of a tuple <*Pr,lc,q*>, where *Pr* is a condition, *lc* is a required statement in program code and *t* is a variable. Conditioned slicing puts away those chunks of original program that cannot affect variables at required statement upon satisfaction of conditions. A conditioned slice is computed in two steps: first, the program is simplified with respect to condition provided in slicing criterion. Thus, statements, not satisfying given condition, are removed. Then, a slice is computed on the reduced program. The reduced program is referred to as a conditioned program. More details on conditioned slicing can be obtained in (Canfora et al., 1998; Cheda et al., 2008; Danicic et al., 2000; (Danicic et al., 2004; Fox et al., 2004; Harman et al., 2001; Hierons et al., 2002).

Motivation

Static slicing does not take into account the information about execution state of the program code. Thus, static slices are constructed irrespective of the input to the program code. Dynamic slicing utilizes the complete information about execution behavior of the program. Thus, dynamic slices are dependent on input to program code. There must be a slicing technique that preserves the execution behavior of the program and is independent of input to the program. Conditioned slicing solves this problem by computing the slices at a particular program point for a variable with respect to a condition. Nowadays, most of the programs are interprocedural in nature. There is hardly any work done on conditioned slicing of interprocedural programs. This paper demonstrates a technique to find out conditioned slices of programs containing multiple procedures.

Objectives

The objective of this work is to propose an algorithm to determine conditioned slices of interprocedural programs using an intermediate representation, a dependence graph. The authors also aim at computing slice time for various programs of different lines of code.

The structure of the rest of paper is done as per following ways. Section 2 delivers some background details of the proposed technique. In Section 3, literature survey is discussed. In Section 4, the proposed approach, i.e., Node-Marking Conditioned Slicing (NMCS) algorithm for interprocedural programs is discussed. Section 5 outlines complexity analysis of NMCS algorithm. In Section 6, the correctness of NMCS algorithm is established. Section 7 provides the implementation and experimental results of the proposed technique. Section 8 provides conclusions and future works.

2. BACKGROUND

This section discusses some basic concepts required to understand the proposed work.

2.1. System Dependence Graph (SDG)

Several researchers have contributed towards the area of program slicing. For a given slicing criterion, a slice can be determined manually for a simple program with less complexity. But, with increasing size and complexity of the programs, automatic slice computation is of

greatest importance. Current automated slicing techniques require that the information available in a program source code be first transformed into some intermediate representation and then the slicing technique be applied. The different types of program representations include control flow graph (CFG), program dependence graph (PDG), system dependence graph (SDG). Details of program representations can be discovered in (Binkley & Gallagher, 1996; Horwitz et al., 1990; Mohapatra, 2005).

3. LITERATURE SURVEY

The technique used by Weiser involved solving data flow equations, and the slice was called static backward slice (Weiser, 1981). Ottenstein and Ottenstein developed program dependence graph (PDG) for intraprocedural programs and computed the slice by traversing backward on the PDG (Ottenstein & Ottenstein, 1984). Horwitz et al. worked out System Dependence Graph (SDG) to symbolize interprocedural program source codes and recommended a two-phase algorithm to calculate slices (Horwitz et al., 1990). The technique of dynamic slicing was first developed by Korel and Laski (Korel & Laski, 1988). Silva performed a survey on some work done on program slicing (Silva, 2012). He reported various features and applications of each technique using examples and established the relations between them.

Canfora et al. brought in the concept of conditioned slicing and developed a general framework, employing subsume relation, for program slicing models that were based on deleting statements (Canfora et al., 1998). A conditioned program slicer, ConSIT, was originated by Danicic et al. (Danicic et al., 2000). Fox et al. put in theory, design, implementation and utility of ConSIT system (Fox et al., 2004). Lucia discussed different slicing methods that were based on deleting statements together with their applications to software engineering (Lucia, 2001). Harman et al. introduced the pre/post conditioned slicing method that could be employed for program analysis regarding pre- and post- conditions (Harman et al., 2001). Hierons et al. discussed utility of conditioned slicing to aid partition testing (Hierons et al., 2002). Danicic et al. proposed an approach to compute executable union slices using conditioned slicing (Danicic et al., 2004). Cheda et al. demonstrated a technique for calculating conditioned slices that were to be employed to first-order functional logic languages (Cheda et al., 2008).

All the works (Canfora et al., 1998; Cheda et al., 2008; Danicic et al., 2000; Danicic et al., 2004; Fox et al., 2004; Harman et al., 2001; Hierons et al., 2002) are concerned with computation of conditioned slices for intraprocedural programs i.e. for programs containing only a single procedure. The interprocedural aspects have not been taken into consideration by them. The work accomplished for calculating conditioned slices for interprocedural programs i.e. for programs containing multiple procedures is also limited. Nowadays, most of the programs contain multiple procedures. So, there is a requirement to find out conditioned slices for programs containing multiple procedures. Sahu et al. proposed an approach to find out conditioned slices for interprocedural programs (Sahu et al., 2016). They had not performed any experimental results analysis and did not take any case studies to test their approach. This paper proposes a method to find out conditioned slices for interprocedural programs along with experimental result analysis. In the absence of any existing related work, the proposed method cannot be compared with any existing work. Table 1 summarizes various slicing approaches developed by different researchers.

4. NODE-MARKING CONDITIONED SLICING (NMCS) ALGORITHM

In this section, an algorithm termed Node-Marking Conditioned Slicing (NMCS) algorithm is proposed for computing conditioned slices of a given problem along with the construction of system dependence graph (SDG).

Conditioned slices can efficiently be calculated by employing a system dependence graph (SDG) as the intermediate program symbolization. Construction of SDG is comprised of the following steps:

1. Constructing Procedure Dependence Graph (PDG) for each method in a class.
2. Constructing System Dependence Graph (SDG) by combining all PDGs.

The algorithm for constructing SDG is provided in Algorithm 1. This paper uses the terms *node* and *vertex* interchangeably.

Table 1. Comparison of various slicing approaches at a glance

Sl. No.	Year	Author(s)	Computed static slice?	Computed dynamic slice?	Computed conditioned slice?	Computed Intraprocedural slice?	Computed Interprocedural slice?	Considered Object-Oriented Features?
1	1981	Weiser (1981)	Yes	No	No	Yes	No	No
2	1984	Ottenstein and Ottenstein (1984)	Yes	No	No	Yes	No	No
3	1988	Korel and Laski (1988)	No	Yes	No	Yes	No	No
4	1990	Horwitz et al. (1990)	Yes	No	No	No	Yes	No
5	1996	Larsen and Harrold (1996)	No	Yes	No	No	Yes	Yes
6	1998	Canfora et al. (1998)	No	No	Yes	Yes	No	No
7	2000	Danicic et al. (2000)	No	No	Yes	Yes	No	No
8	2001	Harman et al. (2001)	No	No	Yes	Yes	No	No
9	2002	Hierons et al. (2002)	No	No	Yes	Yes	No	No
10	2004	Danicic et al. (2004)	No	No	Yes	Yes	No	No
11	2004	Fox et al. (2004)	No	No	Yes	Yes	No	No
12	2005	Mohapatra (2005)	No	Yes	No	No	Yes	Yes
13	2006	Mohapatra et al. (2006)	No	Yes	No	No	Yes	Yes
14	2007	Sahu and Mohapatra (2007)	No	Yes	No	No	Yes	Yes
15	2008	Cheda et al. (2008)	No	No	Yes	Yes	No	No
16	2012	Ray et al. (2012)	No	Yes	No	No	Yes	Yes
17	2013	Ray et al. (2013)	No	Yes	No	No	Yes	Yes
18	2014	Singh et al. (2014)	No	Yes	No	No	Yes	Yes
19	2015	Munjal et al. (2015)	No	Yes	No	No	Yes	Yes
20	2015	Sahu et al. (2015)	No	Yes	No	No	Yes	Yes
21	2017	Proposed approach	No	No	Yes	No	Yes	No

Algorithm 1: SDG Construction Algorithm Input: A program *P*.
Output: The corresponding SDG. Procedure *ConstructPDG()*

```
For start of a method do Make method entry node.
End For
For every executable statement in the program code P do Make a
vertex in the graph.
End For
For all nodes created do
If vertex e is under scope of node d then
Insert a control dependence edge from d to e, d → e. End If
If vertex d controls execution of vertex e, then
Insert a control dependence edge from d to e, d → e. End If
If vertex d defines a variable t and vertex e utilizes t, then
Insert a data dependence edge from d to e, d → e.
End If End For
End Procedure
Procedure ConstructSDG()
For all methods in a class do Call ConstructPDG.
End For
For every parameter present in method call Create an actual-in
```

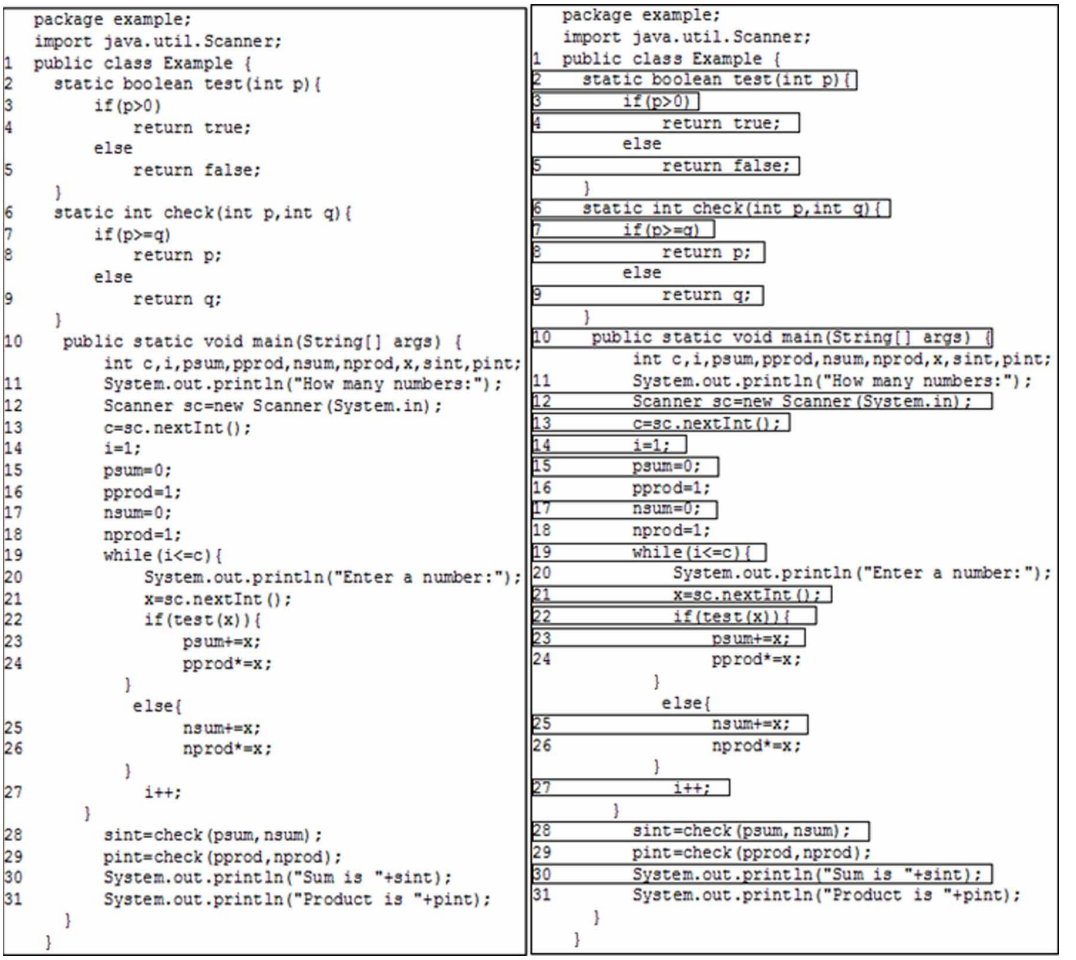
```
parameter vertex.
End For
For every parameter present in method definition Create a formal-
in parameter vertex.
End For
For every parameter in method call that is modified inside the
method Create an actual-out parameter vertex.
End For
For every actual-out parameter vertex
Create corresponding formal-out parameter node. End For
If a method is called vertex d and vertex e defins that method,
then Insert a call edge from d to e, d → e.
Label the edge d → e with d. End If
If vertex e returns a value to call vertex d, then Insert a data
dependence edge from e to d, e → d.
Label the edge e → d with d. End If
If vertex e is actual-in and vertex f is formal-in parameter
vertices for a call vertex d, then Insert a parameter-in edge from
e to f, e → f.
Label the edge e → f with d. End If
If vertex e is actual-out and vertex f is formal-out parameter
vertices for a call vertex d, then
Insert a parameter-out edge from f to e, f → e. Label the edge f →
e with d.
End If
If a path exists from actual-in vertex d to corresponding actual-
out vertex e, then Insert summary edge from d to e, d → e.
End If
If a path exists from actual-in vertex d to corresponding call
vertex e, then Insert summary edge from d to e, d → e.
End If End For
End Procedure
```

An example Java program is depicted in Figure 1. The program is adopted from (Canfora et al., 1998). The program in (Canfora et al., 1998) is written in C. The program used in this paper is written in simple Java and it does not take the object-oriented features into account. Static slice for slicing criterion <*30, sint*> of example program depicted in Figure 1 is comprised of statements numbered 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 17, 19, 21, 22, 23, 25, 27, 28, 30. These statements are also depicted in rectangular boxes in Figure 1 (b). Dynamic slice for slicing criterion <*{c = 3, x = { 8, -3, 11}}, 30, sint*> of example program depicted in Figure 1 is comprised of statements numbered 2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 14, 15, 17, 19, 21, 22, 23, 25, 27, 28, 30. These statements are also depicted in rectangular boxes in Figure 2 (a).

Algorithm 1 is used to construct SDG for example program code. Vertices that satisfy specified condition are marked and conditioned slices are calculated throughout marking process by NMCS algorithm. Corresponding to a method call, the label of the call edge is recorded in a variable, *CSe*. A stack, *SCC*, is maintained to record calling context. When a parameter-in edge is passed through, label of that edge is pushed onto stack, and when a parameter-out edge is passed through, stack is popped. The popped item of stack is compared with *CSe* for equality and conditioned slice is modified if both are equal.

Let *cond_slice(u)* symbolizes conditioned slice for slicing criterion <*Pr, lc, q*>, for *Pr* being a condition, *q* being a variable and *lc* being the statement corresponding to vertex *w*. Let s_1, s_2, …,

Figure 1. (a) An Example Java Program (b) Static w.r.t slicing criterion <30, sint> of example program depicted in Figure 1



```
   package example;
   import java.util.Scanner;
1  public class Example {
2    static boolean test(int p){
3        if(p>0)
4            return true;
         else
5            return false;
     }
6    static int check(int p,int q){
7        if(p>=q)
8            return p;
         else
9            return q;
     }
10    public static void main(String[] args) {
          int c,i,psum,pprod,nsum,nprod,x,sint,pint;
11        System.out.println("How many numbers:");
12        Scanner sc=new Scanner(System.in);
13        c=sc.nextInt();
14        i=1;
15        psum=0;
16        pprod=1;
17        nsum=0;
18        nprod=1;
19        while(i<=c){
20            System.out.println("Enter a number:");
21            x=sc.nextInt();
22            if(test(x)){
23                psum+=x;
24                pprod*=x;
              }
              else{
25                nsum+=x;
26                nprod*=x;
              }
27              i++;
        }
28        sint=check(psum,nsum);
29        pint=check(pprod,nprod);
30        System.out.println("Sum is "+sint);
31        System.out.println("Product is "+pint);
     }
   }
```

```
   package example;
   import java.util.Scanner;
1  public class Example {
2    static boolean test(int p){
3        if(p>0)
4            return true;
         else
5            return false;
     }
6    static int check(int p,int q){
7        if(p>=q)
8            return p;
         else
9            return q;
     }
10    public static void main(String[] args) {
          int c,i,psum,pprod,nsum,nprod,x,sint,pint;
11        System.out.println("How many numbers:");
12        Scanner sc=new Scanner(System.in);
13        c=sc.nextInt();
14        i=1;
15        psum=0;
16        pprod=1;
17        nsum=0;
18        nprod=1;
19        while(i<=c){
20            System.out.println("Enter a number:");
21            x=sc.nextInt();
22            if(test(x)){
23                psum+=x;
24                pprod*=x;
              }
              else{
25                nsum+=x;
26                nprod*=x;
              }
27              i++;
        }
28        sint=check(psum,nsum);
29        pint=check(pprod,nprod);
30        System.out.println("Sum is "+sint);
31        System.out.println("Product is "+pint);
     }
   }
```

s_k indicate all predecessor vertices of w in dependence graph that are marked. So, conditioned slice for slicing criterion <*Pr, lc, q*> is provided as cond_slice(u) = {u, s_1, s_2,…, s_k} U cond_slice(s_1) U cond_slice(s_2) U… U cond_slice(s_k)

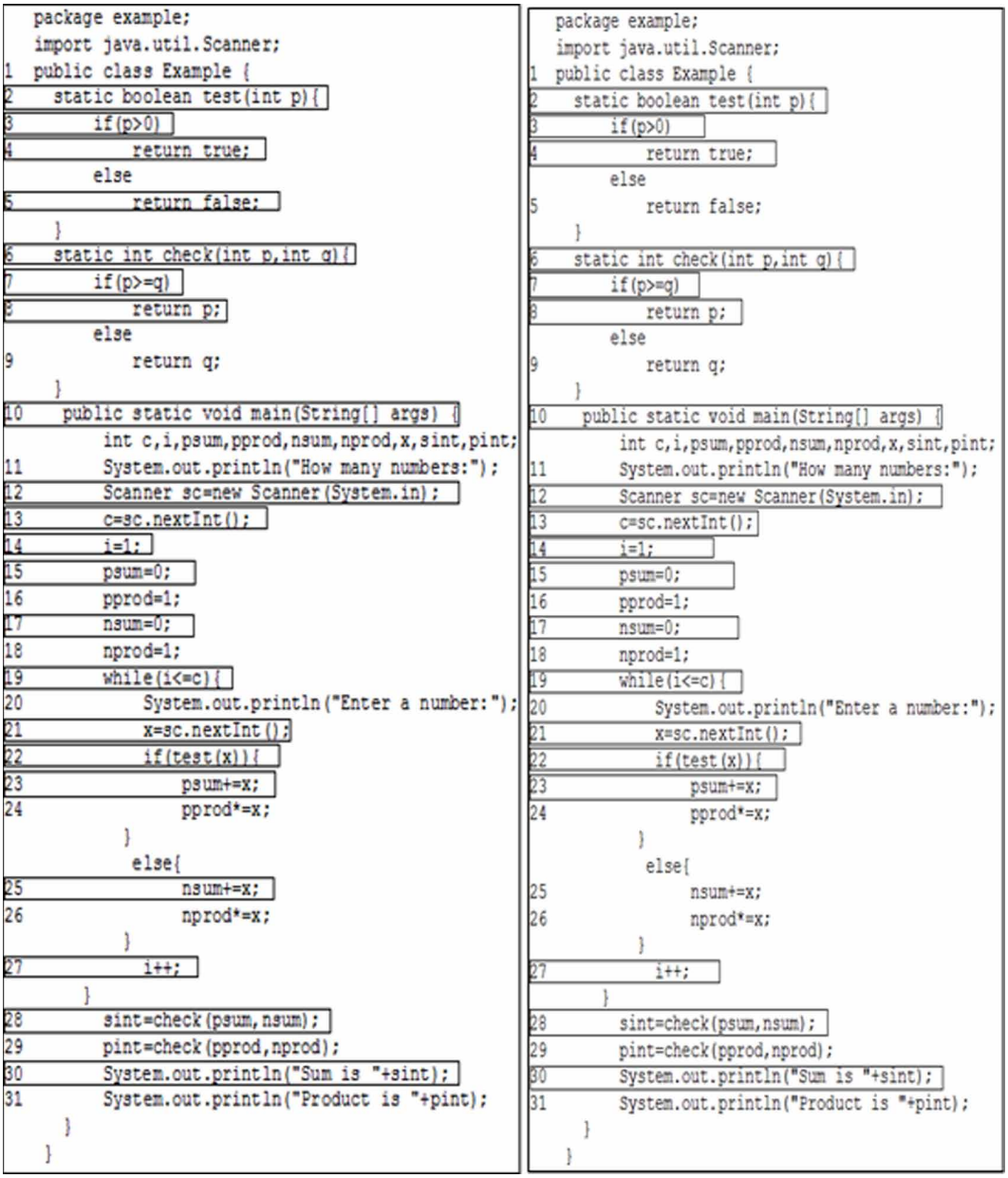
Algorithm 2 presents the proposed NMCS algorithm in pseudocode form. Table 2 depicts the notations used in Algorithm 2.

Algorithm 2: Node-Marking Conditioned Slicing (NMCS) Algorithm Input: SDG of program *P* and the slicing criterion <*Pr,lc,q*>.

Output: List of nodes contained in the required conditioned slice.

1. Set *Marked=φ.* //Initially unmark all nodes.
2. Set *cond_slice(w)= φ*.
3. Check the program *P* for condition *Pr*.
4. For each statement satisfying condition, *Pr*, do
   (a) *Marked=Marked U {w}*. //Mark node *w*.
   (b) Update *cond_slice(w)={w, s_1, s_2, …, s_k} U cond_slice(s_1) U cond_slice(s_2) U… U cond_slice(s_k).*

Figure 2. (a) Dynamic slice with respect to slicing criterion <{n=3,x={8,-3,11}},30,sint> of example program code depicted in Figure 1 (b) Conditioned slice for slicing criterion <{test(x)! = 0}, 30, sint> of example program code depicted in Figure 1



```
   package example;
   import java.util.Scanner;
1  public class Example {
2    static boolean test(int p){
3        if(p>0)
4            return true;
         else
5            return false;
     }
6    static int check(int p,int q){
7        if(p>=q)
8            return p;
         else
9            return q;
     }
10    public static void main(String[] args) {
          int c,i,psum,pprod,nsum,nprod,x,sint,pint;
11        System.out.println("How many numbers:");
12        Scanner sc=new Scanner(System.in);
13        c=sc.nextInt();
14        i=1;
15        psum=0;
16        pprod=1;
17        nsum=0;
18        nprod=1;
19        while(i<=c){
20            System.out.println("Enter a number:");
21            x=sc.nextInt();
22            if(test(x)){
23                psum+=x;
24                pprod*=x;
              }
              else{
25                nsum+=x;
26                nprod*=x;
              }
27            i++;
          }
28        sint=check(psum,nsum);
29        pint=check(pprod,nprod);
30        System.out.println("Sum is "+sint);
31        System.out.println("Product is "+pint);
      }
    }
```

```
   package example;
   import java.util.Scanner;
1  public class Example {
2    static boolean test(int p){
3        if(p>0)
4            return true;
         else
5            return false;
     }
6    static int check(int p,int q){
7        if(p>=q)
8            return p;
         else
9            return q;
     }
10    public static void main(String[] args) {
          int c,i,psum,pprod,nsum,nprod,x,sint,pint;
11        System.out.println("How many numbers:");
12        Scanner sc=new Scanner(System.in);
13        c=sc.nextInt();
14        i=1;
15        psum=0;
16        pprod=1;
17        nsum=0;
18        nprod=1;
19        while(i<=c){
20            System.out.println("Enter a number:");
21            x=sc.nextInt();
22            if(test(x)){
23                psum+=x;
24                pprod*=x;
              }
              else{
25                nsum+=x;
26                nprod*=x;
              }
27            i++;
          }
28        sint=check(psum,nsum);
29        pint=check(pprod,nprod);
30        System.out.println("Sum is "+sint);
31        System.out.println("Product is "+pint);
      }
    }
```

(c) If *w* is a method call vertex then

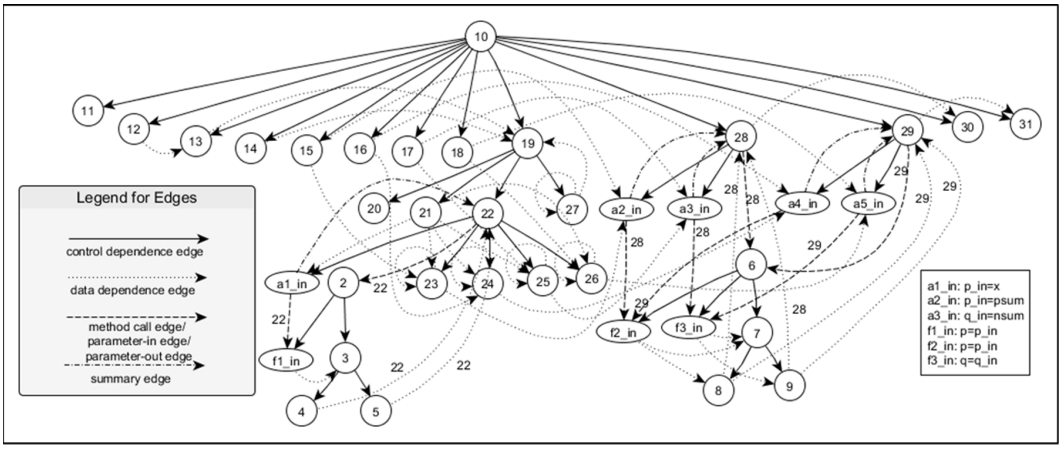
i. $panode_M$=f(M, panode).
ii. Me_M=g(M, Me).
iii. $pfnode_M$=h(M, pfnode).
iv. *Marked=Marked U {w}*. //Mark node *w*.
v. *Marked=Marked U* $panode_M$. //Mark associated actual parameter nodes.
vi. *Marked=Marked U* $\{Me_M\}$. //Mark corresponding method entry node.

Table 2. Notations used in Algorithm 2

Notation	Meaning
Pr	A condition
w	A node in SDG corresponding to a statement *s* in program *P* that satisfies condition *Pr*.
cond_slice(w)	Conditioned slice corresponding to node *w*.
$s_1,s_2,\ldots,s_k$	Predecessor nodes of *w* in SDG that are marked.
U	Set Union Operator. Example: If *X* and *Y* are two sets (or lists), then *X U Y* combines all the elements of set (or list) all the elements of set (or list) *Y* with set *X* (or list).
Label(i,j)	Label of edge *(i,j)*.
Marked	Worklist containing all marked nodes.
panode	Worklist containing actual-in and actual-out parameter vertices associated with a method call or constructor call node.
pfnode	Worklist containing formal-in and formal-out parameter nodes belonging to an entry node of a method corresponding to a method call or constructor call.
Me	Worklist containing all method entry nodes corresponding to method call, constructor call.
Me_M	Worklist containing method entry node of a method *M*.
CSe	Call site variable for edge *e*.
SCC	Stack to keep track of call context.
$panode_M$	Worklist containing actual parameter nodes associated with a method *M* corresponding to method call, constructor call.
$pfnode_M$	Worklist containing formal parameter nodes associated with a method *M*.
f	Function that extracts actual-in and actual-out parameter vertices associated with call of a method *M*.
g	Function that extracts formal-in and formal-out parameter nodes belonging to a method *M*.
h	Function that extracts method entry node of a method *M*.

vii. *Marked=Marked U* $pfnode_M$. //Mark associated formal parameter nodes.
viii. Set CS_e=*w*. //Set call site to current call method *u*.
(d) If *w* is a new operator vertex then
i. $panode_M$=f(M, panode).
ii. Me_M=g(M, Me).
iii. $pfnode_M$=h(M, pfnode).
iv. *Marked=Marked U {w}*. //Mark node *w*.
v. *Marked=Marked U* $panode_M$. //Mark associated actual parameter nodes.
vi. *Marked=Marked U {* Me_M *}*. //Mark corresponding constructor entry node.
vii. *Marked=Marked U* $pfnode_M$. //Mark associated formal parameter nodes.
viii. Set CS_e=*w*. //Set call site to current call method *w*.
(e) If *(w,z)* is a parameter-in edge then
i. Call *SCC.push(Label(w,z))*. // Push label of edge *(w,z)* onto stack *SCC*.
(f) If *(w,z)* is a parameter-out edge then
i. While *SCC* is not empty do
A. *p=SCC.pop()*. //*p* is the popped item
B. If *p=CSe* then
a. Modify *cond_slice(z)={z} U cond_slice(w)*.

Figure 3. SDG of program code depicted in Figure 1



(g) If a vertex *w* returns a value to a vertex *z* that is a method call vertex then

i. While *SCC* is not empty do

A. *p=SCC.pop().//p* is the popped item

B. If $p=CS_e$ then

a. Modify *cond_slice(z)=cond_slice(z) U cond_slice(w).*

5. For a given slicing command <*Pr,lc,q*> do

(a) Seek out *cond_slice(w)* for variable *q*.

(b) Display *cond_slice(w).*

4.1. Working of NMCS Algorithm

An example is employed to explain working of NMCS algorithm. The example Java code depicted in Figure 1 is considered here. Figure 3 depicts its SDG.

All nodes of SDG are first unmarked and *cond_slice=φ* is set for each node *u* of SDG that satisfies the condition *Pr*. Let the slicing criterion be <*{test(x)!=0}, 30, sint*>. The NMCS algorithm first marks nodes 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 2, 3, 4, 23, 24, 27, 28, 6, 7, 8, 9, 29, 6, 7, 8, 30, 31 in order since these vertices satisfy the given condition. Actual parameter vertices associated with a calling method and formal parameter vertices associated with corresponding called method are also marked. During the marking process, conditioned slice for every node in SDG that are marked is also found out.

When node 22, which is a method call node, is marked, CS_e=22 according to Step 4(c)(viii). When node *a1_in* is marked, it is found that the edge *(22→a1_in, 2→f1_in)* is a parameter-in edge. According to Step 4(e)(i), the label of the edge *(22→a1_in, 2→f1_in)*, which is 22, is pushed onto the stack *SCC*. When node 4 is marked, it is found that node 4 returns value *true* to the method call node 22. Thus, according to Step 4(g), the stack *SCC* is repeatedly popped and checked to see if the popped item is equal to CS_e. It is seen that the popped item is *p*=22 and $p=CS_e$. Then, *cond_slice(22)* is updated as *cond_slice(22) U cond_slice(4).*

Similar procedures i.e., Step 4(c)-Step 4(g) are followed for method call vertices 28 and 29. Table 3 depicts the working of the steps 4(c)-4(g) in Algorithm 2 during marking of method call nodes.

Now, conditioned slice for slicing criterion <*{test(x)! = 0}, 30, sint*> i.e., for variable *sint* at statement number 30 with condition that *test(x)* is positive is to be determined. As per NMCS algorithm, conditioned slice is calculated as follows:

Table 3. Working of the steps 4(c)-4(g) in Algorithm 2 during marking of method call nodes

Marked node	Parameter-in edge	Parameter-out edge	CSe	Pushed item	Popped item (p)	Stack content s	Whether p=CSe?	Whether update performed?
22	(22→a1_i n, 2→f1_in)	-	22	22	-	22	-	-
4	-	(4→22)	22	-	22	-	Yes	-
28	(28→a2_i n, 6→f2_in)	-	28	28	-	28	-	-
	(28→a3_in, 6→f3_in)	-	28	28	-	28, 28	-	-
8	-	(8→28)	28	-	28	28	Yes	Yes
			28	-	28	-	Yes	Yes
29	(29→a4_i n, 6→f2_in)	-	29	29	-	29	-	-
	(29→a5_i n, 6→f3_in)	-	29	29	-	29, 29	-	-
8	-	(8→29)	29	-	29	29	Yes	Yes
			29	-	29	-	Yes	Yes

cond_slice(30)={30,10,28} U cond_slice(10) U cond_slice(28). cond_slice(10)={10}.
cond_slice(28)={8, 10, 28, 28→a2_in, 28→a3_in} U cond_slice(8) U cond_slice(10) U cond_slice(28→a2_in) U cond_slice(28→a3_in).
cond_slice(8)={7, 8, 6→f2_in} U cond_slice(6→f2_in) U cond_slice(7).
cond_slice(28→a2_in)={28→a2_in, 15, 23, 28} U cond_slice(15) U cond_slice(23) U cond_slice(28).
cond_slice(28→a3_in)={28→a3_in, 17, 25, 28} U cond_slice(17) U cond_slice(25) U cond_slice(28).

In this way, evaluating all the expressions recursively, the final conditioned slice at statement 30 is obtained as the set of statements corresponding to following set of nodes:

{2, 3, 4, 6, 7, 8, 10, 12, 13, 14, 15, 17, 19, 21, 22, 23, 27, 28, 30}

The bold nodes in Figure 4 represent statements included in conditioned slice. These statements are also depicted in rectangular boxes in Figure 2 (b).

Table 4 depicts complete list of *cond_slice(u)* for each node satisfying condition

Pr=”test(x)!=0”.

5. COMPLEXITY ANALYSIS

This section discusses space and time complexity of the proposed NMCS algorithm.

5.1. Space complexity

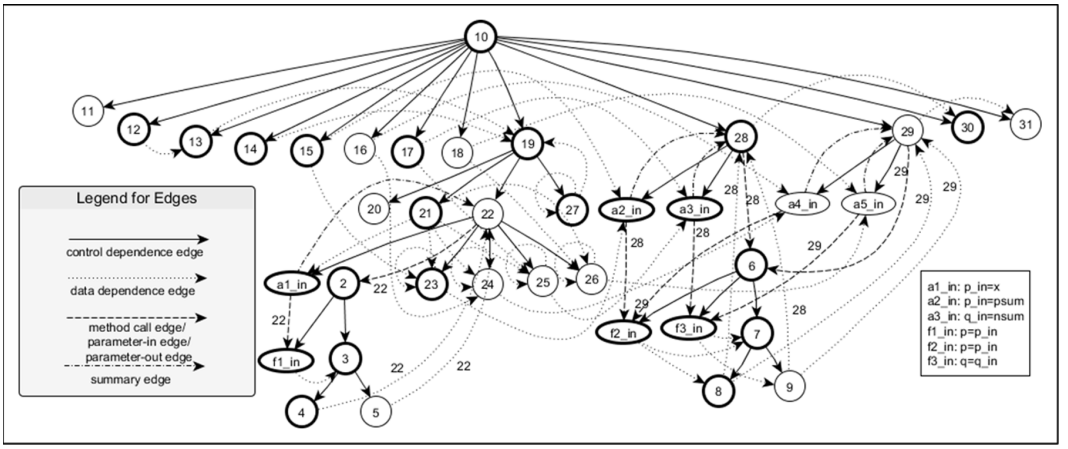
Suppose *IP* is a program having *n* statements. A single vertex in the SDG will be used to represent each statement. But, the statements representing method invocation and method definition will require extra vertices to take care of representing the actual and formal parameters. For such a statement, number of additional vertices required is same as number of the actual or formal parameters. Let the

Table 4. List of *cond_slice(u)* for each node satisfying condition *Pr="test(x)!=0"*

w	*cond_slice(w)*
2	2, 10, 12, 13, 14, 19, 21, 22, 27
3	2, 3, 10, 12, 13, 14, 19, 21, 22, 27
4	2, 3, 4, 10, 12, 13, 14, 19, 21, 22, 27
6	2, 3, 4, 6, 10, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 27, 28, 29
7	2, 3, 4, 6, 7, 10, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 27, 28, 29
8	2, 3, 4, 6, 7, 8, 10, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 27, 28, 29
10	10
11	10, 11
12	10, 12
13	10, 12, 13
14	10, 14
15	10, 15
16	10, 16
17	10, 17
18	10, 18
19	10, 12, 13, 14, 19, 27
20	10, 12, 13, 14, 19, 20, 27
21	10, 12, 13, 14, 19, 21, 27
22	2, 3, 4, 10, 12, 13, 14, 19, 21, 22, 27
23	2, 3, 4, 10, 12, 13, 14, 15, 19, 21, 22, 23, 27
24	2, 3, 4, 10, 12, 13, 14, 16, 19, 21, 22, 24, 27
27	10, 14, 19, 27
28	2, 3, 4, 6, 7, 8, 10, 12, 13, 14, 15, 17, 19, 21, 22, 23, 27, 28
29	2, 3, 4, 6, 7, 8, 10, 12, 13, 14, 16, 18, 19, 21, 22, 23, 27, 29
30	2, 3, 4, 6, 7, 8, 10, 12, 13, 14, 15, 17, 19, 21, 22, 23, 27, 28, 30
31	2, 3, 4, 6, 7, 8, 10, 12, 13, 14, 16, 18, 19, 21, 22, 23, 27, 29, 31

following assumption be taken: the number of parameters in a method invocation is less than m, where m is some bounded positive integer. Since, the number of actual or formal parameters in a method call is a small bounded positive number, so m must be a small bounded positive number. It can be stated that at most m number of nodes in SDG are used to represent each statement of program code *IP*. Thus, space requirement for SDG of a program code *IP* containing n statements is $O(mn^2)$. Since m is a small bounded positive number, so, space required for storing the SDG is $O(n^2)$. Again, some amount of space is required for stack that is employed for keeping track of the calling context. This necessitates maximum $O(k)$ space, where k is the number of parameters in a method invocation since the stack is used for storing the labels of only parameter-in edges and the stack is popped out when parameter-out edges are encountered. Also, some amount of space is needed for storing *cond_slice(u)* for each satisfied statement w of program code *IP*. This requires maximum $O(n)$ space. So, for n statements in program code *IP*, maximum $O(n^2)$ space is needed to store *cond_slice(u)*. Thus, total

Figure 4. Bold nodes exhibiting conditioned slice for slicing criterion <test(x)!=0,30,sint> of example Java code given in Figure 1



space required for the NMCS algorithm is $O(n^2)+O(m)$. As k is much less than n, the total space requirement for NMCS algorithm is $O(n^2)$, n being number of statements in program.

5.2. Time complexity

Suppose P is a program code having n statements. The total time complexity of NMCS algorithm is due to following components:

- Time needed to construct SDG which is $O(n^2)$.
- Time needed to store required information at each node of SDG, which is $O(n)$.
- Time needed to traverse SDG and reach at specified node, which is $O(n^2)$.
- Time required to perform push and pop operations on stack, which is $O(1)$.
- Time needed to seek out data structure *cond_slice(u)* for obtaining conditioned slice, which is $O(1)$ as every node of SDG is annotated with its most recent conditioned slice.

Hence, total time requirement for NMCS algorithm to compute conditioned slice is $O(n^2)$.

6. CORRECTNESS PROOF

This section presents correctness proof of NMCS algorithm.

Theorem 1

The conditioned slice computed w.r.t. a given slicing criterion by NMCS algorithm is always correct.

Proof

The method of Mathematical induction has been used to establish the correctness of the proposed NMCS algorithm. Consider a program P for which NMCS algorithm computes a conditioned slice. For a given condition, the conditioned slice w.r.t. first statement that satisfies condition is certainly correct. It can also be argued that conditioned slice with respect to second statement satisfying the given condition is also correct. Suppose the NMCS algorithm has correctly computed the conditioned slices before the statement *lc* representing the node u that satisfies the given condition. It is only required to determine that conditioned slice computed after statement u satisfying given condition is correct. Let Pr be the condition, q be the variable used at u and *cond_slice(u)* be conditioned slice w.r.t. slicing criterion

<*Pr,lc,q*>. It is clear that conditioned slice *cond_slice(u)* is composed of all those statements by which value of variable *q* has been influenced and the condition *Pr* has been satisfied. The proposed NMCS algorithm has marked all the nodes that satisfies the condition *Pr*. Let m_1, m_2, …, m_k be the marked vertices on which the vertex *u* is dependent. Then, *cond_slice(u)={u} U cond_slice*(m_1) *U cond_slice*(m_2) *U … U cond_slice*(m_k). Since *cond_slice*(m_1), *cond_slice*(m_2), …, *cond_slice*(m_k) are all *correct* conditioned slices, the conditioned slice *cond_slice(u)* calculated in Step 4(b) of the algorithm must also be *correct*. Thus, the correctness of NMCS algorithm is established. □

7. IMPLEMENTATION

A tool termed *Conditioned Slicing Tool* (CST) (Figure 5) has been developed to compute the conditioned slices. The architecture of CST is given in Figure 3. CST is composed of two components: *SDG Generator* and *Slicer*. SDG Generator consists of two components: ASM Framework and JSDG Framework. ASM is an open source Java bytecode manipulation and analysis framework. It is employed to manipulate the existing classes and to generate new classes dynamically. Several packages are collected together in ASM for analysis of different tasks. The given Java program is compiled using Java compiler, and the generated Java bytecode is given as input to the ASM framework. The JSDG framework collects all the information related to a program such as type of statement, type of dependency between statements, the number of classes, the number of methods, etc. from ASM framework and generates the SDG of program code. The generated SDG is fed to Slicer component. The slicing criterion is given to Slicer through a Graphical User Interface (GUI). The Slicer uses the SDG and slicing criterion to compute the conditioned slice. The computed slices are fed to GUI for display to the user. The different packages used in CST are provided in Table 5. Different codes utilized for storing different forms of edges in SDG are provided in Table 6.

7.1. Experimental Results

The CST is applied on various programs given in Table 7 with different slicing criteria for several conditions. Since the conditioned slices are computed at different statements of a program, average slice computation time is calculated. The details of the outcomes are given in Table 8. The slice computation time includes the time to find out the conditioned slice after generating SDG and the time to extract slice with respect to a slicing criterion.

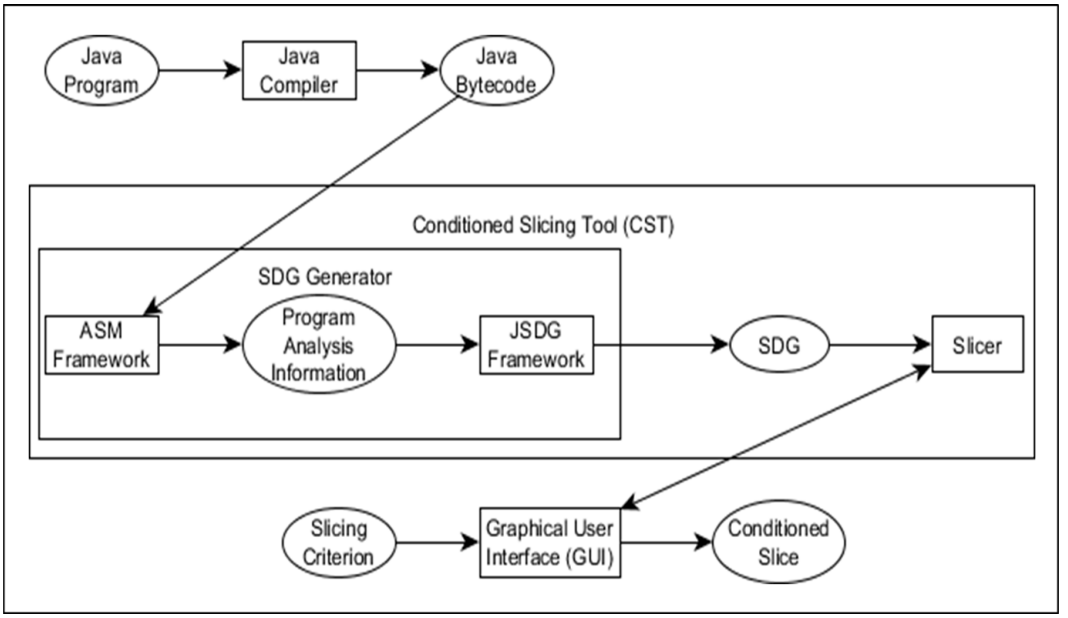
It can be observed from Table 8 that the SDG generation time and slice computation time increases with the increase of lines of code. Also, the slice computation time increases as the number of conditions increases. Figure 6 depicts the SDG generation time and slice computation time for the case studies taken.

7.2. Threats to Validity

This section discusses some possible threats to validity of proposed NMCS algorithm.

- The presence of recursions, polymorphism, exception handling, etc. are not considered in this approach.
- This approach also does not consider multithreading programs and distributed programs. Only single threaded programs have been considered.
- The tool CST takes only Java programs. It may not work well for programs written in C, C++, C# languages.
- The tool CST has been tested on seven moderate sized programs given in Table 4. It may not handle very large size industrial projects as the generated SDG will be very large and unmanageable.

Figure 5. Architecture of Conditioned Slicing Tool (CST)



8. CONCLUSION AND FUTURE WORKS

This paper has demonstrated a technique to find out conditioned slices for interprocedural programs i.e. programs containing multiple procedures. The technique has been termed node-marking conditioned slicing (NMCS) algorithm. System dependence graph (SDG) is first constructed. Next, nodes satisfying condition, specified in slicing criterion, are marked. The slices are also found out during the marking process using marked nodes only. In future, the authors will develop some techniques to calculate conditioned slices for large and complex object-oriented software, aspect-oriented software, feature-oriented software, web-based applications, etc. The authors will also extend this work to find out conditioned slices of concurrent and distributed programs.

Table 5. Packages employed in CST

Package	Description
com.asm.internal	Stores the internal classes that operate with ASM framework
com.asm.internal.util	Stores the utility classes that operate with ASM framework
com.graph	Stores the common attributes of a graph and determines the dependencies amongst different parts of program
com.graph.element	Stores the basic element of a graph
com.graph.internal	Stores the internal representation of a graph
com.graph.Iterator	Stores the different iterators for different searching algorithm
com.graph.pdg	Stores the program dependence graph related information
com.graph.sdg	Stores the system dependence graph related information
com.util	Stores the common utility classes
com.util.datastructure	Stores the common data structure classes

Table 6. Encodings employed for different forms of edges in SDG

Code	Edge Type
0	No edge
1	Control dependence edge
2	Data dependence edge
3	Call edge
4	Parameter-in edge
5	Parameter-out edge
6	Summary edge

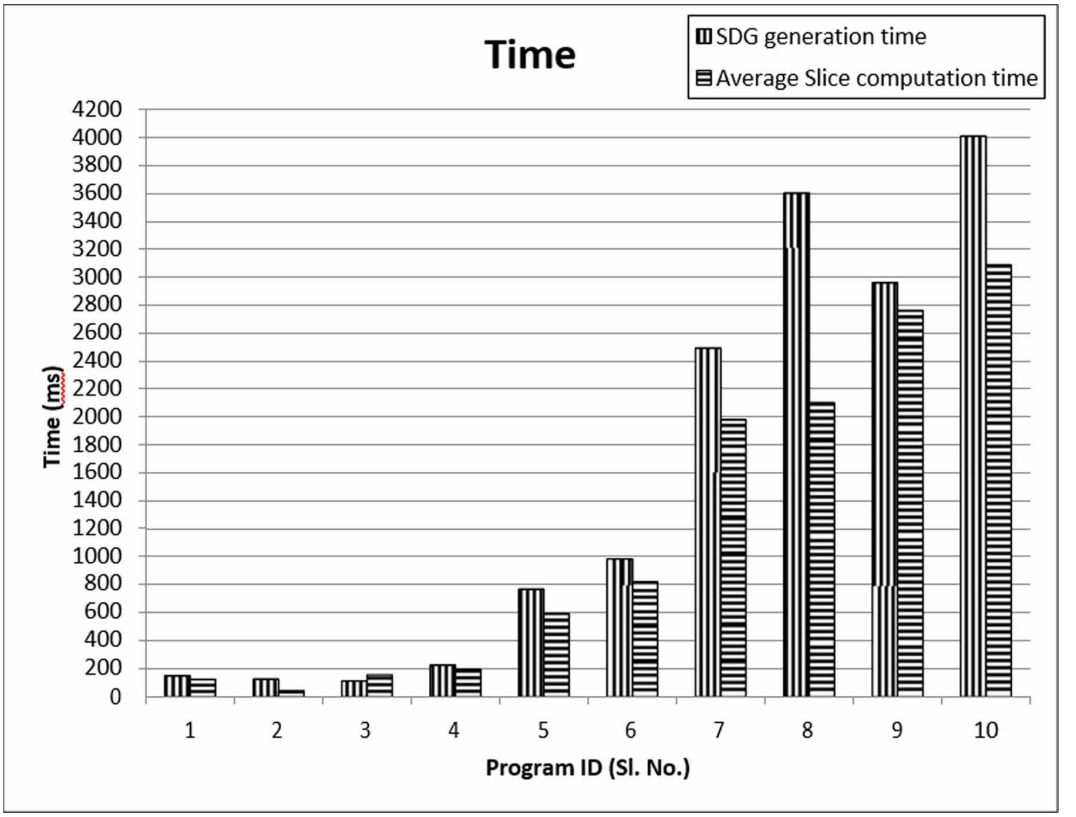
Table 7. Programs used

Sl. No.	Program Name	Description
1	SumProd	Finds the summation and product of positive and negative integers (Figure 2 (a))
2	FindMax	Finds the maximum of three numbers
3	BankApplication	Simulates a simple banking application with deposit and withdrawal feature
4	TopologicalSort	Performs a topological sort on a directed graph
5	CircularSingleLinkedList	Performs creation, insertion, deletion operations on a circular single linked list
6	RedBlackTree	Performs creation, insertion, deletion operations on a Red-Black tree
7	ShortestPath	Simulates Dijkstra's algorithm to determine shortest path between two nodes on a graph
8	BankingSystem	Simulates a simple banking system for transactions on an account
9	Elevator	Simulates an elevator system
10	ATMSimulation	Simulates an ATM system

Table 8. Average Slice computation time

Sl. No.	Program Name	Lines of Code	Number of conditions	Number of nodes in SDG	Number of edges in SDG	SDG generation time (ms)	Average Slice computation time (ms)
1	FindMax	17	1	23	49	127	109
2	SumProd	31	1	37	85	149	124
3	BankApplication	52	2	60	98	173	157
4	TopologicalSort	85	2	94	172	226	195
5	CircularSingleLinkedList	265	3	286	497	770	593
6	RedBlackTree	280	3	315	453	986	823
7	ShortestPath	502	4	573	921	2496	1983
8	BankingSystem	887	5	913	1209	3605	2103
9	Elevator	1089	5	1109	1385	2963	2765
10	ATMSimulation	1283	6	1315	1485	4013	3091

Figure 6. SDG generation time and slice computation time for various case studies



ACKNOWLEDGMENT

Thank you to Durga Prasad Mohapatra for his contribution and help towards the publication of this manuscript.

REFERENCES

Binkley, D. W., & Gallagher, K. B. (1996). Program slicing. *Advances in Computers*, *43*, 1–50. doi:10.1016/S0065-2458(08)60641-5

Canfora, G., Cimitile, A., & De Lucia, A. (1998). Conditioned program slicing. *Information and Software Technology*, *40*(11), 595–607. doi:10.1016/S0950-5849(98)00086-X

Cheda, D., & Cavadini, S. (2008). Conditioned Slicing for First-Order Functional Logic Programs. In *Proceedings of 17th International Workshop on Functional and (Constraint) Logic Programming (WFLP '08)* (pp. 1-14).

Danicic, S., De Lucia, A., & Harman, M. (2004, June). Building executable union slices using conditioned slicing. In *Proceedings. 12th IEEE International Workshop on Program Comprehension* (pp. 89-97). IEEE. doi:10.1109/WPC.2004.1311051

Danicic, S., Fox, C., Harman, M., & Hierons, R. M. (2000, October). ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)* (pp. 216-226).

De Lucia, A. (2001). Program slicing: Methods and applications. In *Proceedings. First IEEE International Workshop on Source Code Analysis and Manipulation 2001* (pp. 142-149). IEEE.

Fox, C., Danicic, S., Harman, M., & Hierons, R. M. (2004). ConSIT: A fully automated conditioned program slicer. *Software, Practice & Experience*, *34*(1), 15–46. doi:10.1002/spe.556

Harman, M., Hierons, R., Fox, C., Danicic, S., & Howroyd, J. (2001, November). Pre/post conditioned slicing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)* (p. 138). IEEE Computer Society.

Hierons, R., Harman, M., Fox, C., Ouarbya, L., & Daoudi, M. (2002). Conditioned slicing supports partition testing. *Software Testing, Verification & Reliability*, *12*(1), 23–28. doi:10.1002/stvr.232

Horwitz, S., Reps, T., & Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, *12*(1), 26–60. doi:10.1145/77606.77608

Korel, B., & Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, *29*(3), 155–163. doi:10.1016/0020-0190(88)90054-3

Larsen, L., & Harrold, M. J. (1996, March). Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering 1996* (pp. 495-505). IEEE.

Mohapatra, D. P. (2005). *Dynamic Slicing of Object-Oriented Programs* [Doctoral dissertation]. Indian Institute of Technology, Kharagpur, Kharagpur

Mohapatra, D. P., Kumar, R., Mall, R., Kumar, D. S., & Bhasin, M. (2006). Distributed dynamic slicing of Java programs. *Journal of Systems and Software*, *79*(12), 1661–1678. doi:10.1016/j.jss.2006.01.009

Munjal, D., Singh, J., Panda, S., & Mohapatra, D. P. (2015, July). Automated Slicing of Aspect-Oriented Programs using Bytecode Analysis. In *2015 IEEE 39th Annual Computer Software and Applications Conference (COMPSAC)* (Vol. 2, pp. 191-199). IEEE. doi:10.1109/COMPSAC.2015.98

Ottenstein, K. J., & Ottenstein, L. M. (1984, April). The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, *19*(5), 177–184. doi:10.1145/390011.808263

Ray, A., Mishra, S., & Mohapatra, D. P. (2012). A Novel Approach for Computing Dynamic Slices of Aspect-Oriented Programs. *International Journal of Computer Information Systems*, *5*(3), 6–12.

Ray, A., Mishra, S., & Mohapatra, D. P. (2013). An Approach for Computing Dynamic Slice of Concurrent Aspect-Oriented Programs. *International Journal of Software Engineering and Its Applications*, *7*(1), 13–32.

Sahu, M., & Mohapatra, D. P. (2007, December). A node-marking technique for dynamic slicing of aspect-oriented programs. In *10th International Conference on Information Technology (ICIT 2007)* (pp. 155-160). IEEE. doi:10.1109/ICIT.2007.70

Sahu, M., & Mohapatra, D. P. (2016). Dynamic slicing of feature-oriented programs. In *Proceedings of 3rd International Conference on Advanced Computing, Networking and Informatics* (pp. 381-388). New Delhi, India: Springer. doi:10.1007/978-81-322-2529-4_40

Sahu, M., & Mohapatra, D. P. (2016). Interprocedural Conditioned Slicing. In *Proceedings of 3rd International Conference on Computational Intelligence in Data Mining (ICCIDM 2016)* (pp. 469-479).

Silva, J. (2012). A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, *44*(3), 12. doi:10.1145/2187671.2187674

Singh, J., Munjal, D., & Mohapatra, D. P. (2014, December). Context sensitive dynamic slicing of concurrent aspect-oriented programs. In *2014 21st Asia-Pacific Software Engineering Conference (APSEC),* (Vol. 1, pp. 167-174). IEEE. doi:10.1109/APSEC.2014.35

Weiser, M. (1981, March). Program slicing. In *Proceedings of the 5th international conference on Software engineering* (pp. 439-449). IEEE Press.

Madhusmita Sahu is an assistant professor of C V Raman Computer Academy, Bhubaneswar. She is pursuing her PhD at National Institute of Technology, Rourkela, Odisha, India. Her research interests include software engineering, software testing, compiler design, data structures and algorithms.