Intra-procedural Slicing Using Program Dependence Table

Reza Mohammadian
Department of computer
Islamic Azad University
Qom, Iran
mohammadian_reza@hotmail.com

Saeed Doostali
Department of computer
University of Kashan
Kashan, Iran
doostali.s@gmail.com

Seyed Morteza Babamir
Department of computer
University of Kashan
Kashan, Iran
babamir@kashanu.ac.ir

***Abstract*—Program slicing as a program analysis technique extracts the statements of a program to restrict the focus of a task to specific sub-components of a program. This technique has many applications in software maintenance such as debugging, regression testing, program understanding and reverse engineering. Most of the researches in program slicing used the different type of dependence graphs as intermediate representation tools and then proposed the slicing algorithms based on it. Drawing the dependence graphs to represent large and complex programs is very complicated and it is impossible in non-automatic approaches that requires the experted user to perform an extensive amount of manual work to apply the proposed approach. Moreover scrolling a dependence graph to find the executable slices of a program is one of the most common issue in software testing. In this paper, we introduce the program dependence table as a new intermediate representation tool. Then, we proposed five automatic intra-procedural slicing algorithms including forward, backward, chop, union and backbone based on the program dependence table.**

***Keywords*—Program slicing, Program dependence graph, Program dependence table, Intra-procedural slicing.**

I. Introduction

Software testing as a vital part of software development is a process of executing a software program with the intent of finding potential errors [1]. This process can be done manually or automatically. Using automated software testing can optimize the software testing lifecycle thus save time and cost [2], [3]. Program slicing is one of the most important techniques which is used to improve the efficiency of software testing [4]. This technique extracts the statements of a program which are relevant to a given computation. In fact program slicing is a feasible method to restrict the focus of a task to specific sub-components of a program [5]. In addition to the software testing, there are more applications of this technique on the various of software engineering activities such as program debugging, maintenance and understanding [6], software measurement [7], cohesion measuring [8], detecting dead code [9], test data generation [10], prevention of state explosion in model checking [11] and so on. In spite of extensive effect of program slicing in various software activities especially testing, it is not very used due to its complexity for large and complex programs [12].

The basic idea of program slicing was first introduced by Weiser in 1981 [13]. In the following, it has been introduced for various programming paradigms including inter and intra procedural [14], [15], object-oriented [16], aspect-oriented [17], feature-oriented [18], functional [19] and the systems which are written in multiple programming languages [20]. Most of the researches in program slicing used the different type of dependence graph as intermediate representation tool and then proposed the slicing algorithms based on it. In [14] the authors utilized program dependence graph to extract the intra-procedural programs. To generate inter-procedural slices the system dependence graph is applied in [21]. Class dependence graph has been used in object-oriented program slicing [16], and An extended version of aspect-oriented system dependence graph was utilized to slice the aspect-oriented programs [17]. Moreover, there are more researches in dynamic slicing. In [22] Singh et al. proposed a dynamic slicing approach for concurrent AspectJ programs based on multi-threaded aspect-oriented dependence graph. In [18] dynamic feature composition dependence graph has been used to create a dynamic slice for a feature-oriented program.

Drawing the dependence graph to slice large and complex programs is very complicated and it is impossible in non-automatic approaches that requires the expert user to perform an extensive amount of manual work to apply the proposed approach [12]. In fact, computing a program slice automatically is the main challenge in program slicing approaches. Moreover, extracting the programs to find the executable slices can help us to test the programs.

In this paper, we will introduce the program dependence table (PDT) as a new intermediate representation tool and five automatic slicing algorithms including forward, backward, chop, union and backbone based on the dependence table. The proposed algorithms can compute static slices of an intra-procedural program based on its PDT.

The layouts of the paper are as follows: In Section II are mentioned the basic concepts of program dependence graph and various types of program slicing including backward, forward, chop, backbone and union. In Section III, the details of PDT for intra-procedural programs are presented. Our proposed slicing algorithms are described in Section IV and at the end Section V concludes the paper with some insights

into our future works.

II. Background

A. Program dependence graph

As we mentioned, program slicing reduces the program to those statements that are relevant for a particular context. To do so, it utilizes the dependency relation between the program statements to identify the statements that affect or are affected by a point of interest, which is called slicing criterion [23]. This point consists of a pair $\langle s, v\rangle$, where s is a program point and v is a subset of program variables. The approach of slicing is based on a Program Dependence Graph (PDG).

A PDG $\mathcal{G} = \langle \mathcal{V}, \mathcal{E}\rangle$ is a directed graph where $\mathcal{V}$ is the statements of a source program and $\mathcal{E}$ is the dependence relations (i.e. data dependence or control dependence) between the statements. An edge $s_i \rightarrow s_j$ where $s_i, s_j \in \mathcal{V}$ represents dependence of the statement s_j on statement s_i.

Definition II.1. Control Dependence: There is a control dependency between the statements s_i and s_j, if s_i is a conditional predicate and execution of s_j is determined by the result of s_i [9].

Definition II.2. Data Dependence: There is a data dependency between the statements s_i and s_j, if for any variable $x \in s_i$ assigns the value to $x \in s_j$ refers to x, and there exists at least one execution path in between s_i and s_j without the value of x being changed [9].

Fig. 2 demonstrates the PDG of Fig. 1.a. In the graph $s_i \xrightarrow{c} s_j$ shows control dependency edge and $s_i \xrightarrow{d} s_j$ denotes data dependency edge. For more clear, we explain this example: since the statement main (i.e. s_0) has control dependence on statements $s_1 - s_7$ and $s_{13} - s_{15}$, therefore a control dependency edge exists from s_0 to $s_1 - s_7$, s_{12} and s_{13}, while statement while (i.e. s_7) has control dependence on $s_8 - s_{12}$, so a direct control edge exists from s_7 to $s_8 - s_{12}$. On the other hand, variable a is defined by statement s_2 and used by s_8, s_9 and s_{13}, so a data dependency edge exists from s_2 to s_8, s_9 and s_{13}.

B. Program slicing

Program slicing can be divided into five types of categories: backward, forward, chop, backbone and union. Briefly, they are the following:

Backward program slicing: In backward slicing, we are interested in all the statements that could influence the slicing criterion, so we traverse the program backward from the slicing criterion. It can help us to find localizing errors. Fig. 1.b and Fig. 1.c show the backward program slices of $\langle s_{13}, q\rangle$ and $\langle s_{14}, sp\rangle$ respectively.

Forward program slicing: In forward slicing, all the statements that could be influenced by the slicing criterion are interested, so a program can be traversed forward from a desired point. It is useful when a statement is modified and we want to check the statements affected by the modification

statement no.	a	b	c
s_0	main(){	main(){	main(){
s_1	int a=0;		
s_2	int q=0;	int q=0;	int q=0;
s_3	int p=1;		int p=1;
s_4	int sp=0;		int sp=0;
s_5	int n=0;	int n=0;	int n=0;
s_6	cin>>n;	cin>>n;	cin>>n;
s_7	while (n>0){	while (n>0){	while (n>0){
s_8	q=q+n;	q=q+n;	q=q+n;
s_9	p=q*q;		p=q*q;
s_{10}	sp=p+sp;		sp=p+sp;
s_{11}	n=n-1;	n=n-1;	n=n-1;
s_{12}	}	}	}
s_{13}	cout<<q;	cout<<q;	
s_{14}	cout<<sp;		cout<<sp;
s_{15}	}	}	}

d	e	f	g
		main(){	main(){
		int q=0;	int q=0;
			int p=1;
			int sp=0;
		int n=0;	int n=0;
		cin>>n;	cin>>n;
		while (n>0){	while (n>0){
q=q+n;	q=q+n;	q=q+n;	q=q+n;
p=q*q;			p=q*q;
sp=p+sp;			sp=p+sp;
		n=n-1;	n=n-1;
		}	}
cout<<q;	cout<<q;		cout<<q;
cout<<sp;			cout<<sp;
		}	}

Fig. 1. Different kind of program slicing, (a) source code, (b) and (c) backward slices of $\langle s_{13}, q\rangle$ and $\langle s_{14}, sp\rangle$ respectively, (d) forward slice of $\langle s_8, q\rangle$, (e) chop of $\langle s_8, q\rangle$ and $\langle s_{13}, q\rangle$, (f) backbone of $\langle s_{13}, q\rangle$ and $\langle s_{14}, sp\rangle$ and (g) union of $\langle s_{13}, q\rangle$ and $\langle s_{14}, sp\rangle$

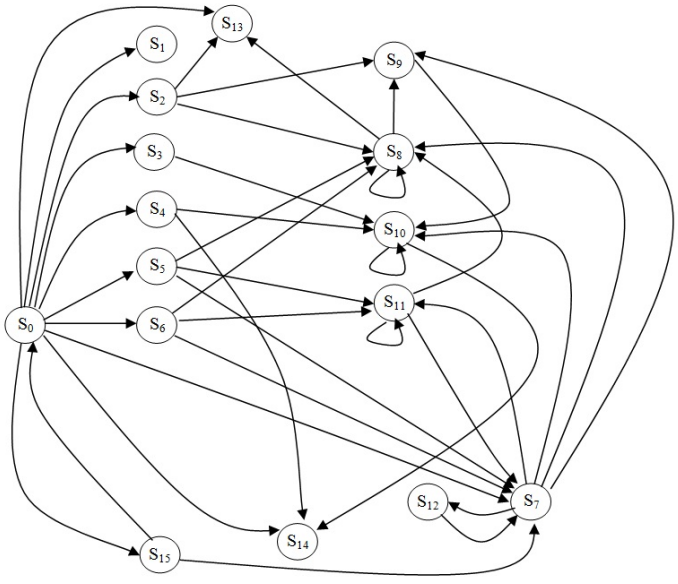

Fig. 2. PDG of Fig. 1.a

which is called ripple analysis. Fig. 1.d demonstrates forward program slice of $\langle s_8, q\rangle$.

Chop: Chop is the intersection of a forward and a backward slice. Fig. 1.e shows a chop for the forward program slice $\langle s_8, q\rangle$ and the backward program slice $\langle s_{13}, q\rangle$. Chops can help us to discover how one statement influences another statement.

Backbone: Backbone is the intersection of two backward slices. Fig. 1.f shows a backbone for $\langle s_{13}, q\rangle$ and $\langle s_{14}, sp\rangle$. Backbones are useful for discovering the parts of an application that contribute to the computation of several values [24].

Union slicing: The union of backward slices for all output variable of a program is called union slicing. Fig. 1.g denotes the union slicing for the program of Fig. 1.a. Clearly, this slicing includes all the statements of backward slices $\langle s_{13}, q\rangle$ and $\langle s_{14}, sp\rangle$. A union slice can help us to find the dead codes. For example, the union slice in Fig. 1.g does not include s_1, hence this statement can not effect on the outputs and it is a dead code.

On the other side, we can divide program slicing into two categories: dynamic and static. A dynamic program slice is a subset of the program that is entirely defined on the basis of a computation, i.e., it depends on the specific value for the desired variable. While a static program slice is generated from program structure [25].

III. Program dependence table

Program Dependence Table (PDT) is a $m \times m$ matrix where m is the number of program statements. PDT is created by the algorithm 1.

Our algorithm takes s_code as the input source code and returns its PDT. The algorithm uses two initially empty global stack, called DS and LS which store all dependencies between the statements and dependencies between the loop statements (e.g. for, while, etc) respectively. We also define a list for each variable x in the source code to store its def-statements (we call it $L_{def}(x)$).

Definition III.1. def-statement: If the variable x is defined in the statements s_i then s_i is called def-statement of x.

Now we describe our encoding for each statement in the source code such as $s_i \in \mathcal{V}$: (a) if the stack DS is not empty then we set $PDT[tos(DS), i]$ to c, where $tos(DS)$ returns first element of DS located on the top of the stack. This means that, there is a control dependency between s_i and $tos(DS)$. (b) for any used variable $x \in s_i$, we set $PDT[L_{def}(x), i]$ to d, i.e. there is a data dependency between s_i and all def-statements of x. (c) For each defined variable in s_i such as x, we add i to $L_{def}(x)$. (d) If s_i is a type of loop statements, then we push i on the stack LS. (e) If s_i contains $\{$, we push i on the stack DS. (f) If s_i contains $\}$, then we set $PDT[i, tos(DS)]$ to c. Next, if $tos(DS) = tos(LS)$, then we check the equality of first and second elements of the stack LS. If they are equal, then LS will be popped twice, otherwise we jump from s_i to the line $tos(LS)$ of source code. finally, the top of DS will be popped off. The reason of doing steps (e) and (f) is making the executable slices.

Algorithm 1 Creating PDT

```
1:  for all s_i ∈ s_code do
2:    if not(empty(DS)) then
3:      PDT[tos(DS), i] ← c
4:    end if
5:    X_used ← all_used_variables(s_i)
6:    for all x ∈ X_used do
7:      PDT[L_def(x), i] ← d
8:    end for
9:    X_def ← all_defined_variables(s_i)
10:   for all x ∈ X_def do
11:     L_def(x) ← L_def(x) ∪ i
12:   end for
13:   if s_i is a loop statement then
14:     push i on LS
15:   end if
16:   if s_i contains { then
17:     push i on DS
18:   end if
19:   if s_i contains } then
20:     PDT[i, tos(DS)] ← c
21:     if tos(DS) = tos(LS) then
22:       if tos(LS) = sos(LS) then
23:         pop LS
24:         pop LS
25:       else
26:         jump to s_tos(LS)
27:       end if
28:     end if
29:     pop DS
30:   end if
31: end for
32: return PDT
```

For more clear, we explain our algorithm for the program of Fig. 1.a. Since the stack DS is empty when we encounter the statement s_0 and also no variable has been used in this statement, so PDT does not change. Moreover, s_0 is not a def-statement, hence no element is added to the def-statement-lists. Statement s_0 has $\{$, so we push 0 on the stack DS (see row 2 in Table I).

For the statement s_1, top of DS is 0, so $PDT[0, 1]$ is set to c. The variable a has been defined in s_1, thus we add 1 to the def-statement-list $L_{def}(a)$. Other conditions are not satisfied in this statement (see row 3 in Table I). Statements $s_2 - s_5$ are encoded in a similar way.

When we encounter s_6, top of the stack DS is 0, thus $PDT[0, 6] = c$. Since the variable n has been used in s_6 and $L_{def}(n) = \{5\}$, hence $PDT[5, 6]$ is set to d. Further, s_6 defines the variable n, so we add 6 to the def-statement-list $L_{def}(n)$ (see row 8 in Table I).

For the statement s_7, since $tos(DS) = 0$ thus we set $PDT[0, 7]$ to c. This statement uses the variable n, so

$PDT[L_{def}(n), 7]$ is set to d where the def-statement-list of n contains 5 and 6. Moreover, we push the number of this statement on DS and LS because s_7 has { and it is a loop statement (see row 9 in Table I).

Top of the stack DS is 7 when we encounter s_8, thus $PDT[7, 8] = c$. The variables n and q have been used in s_8, so the proposed algorithm assigns the data dependency between s_8 and the def-statements of n and q. Since $L_{def}(n) = \{5, 6\}$ and $L_{def}(q) = \{2\}$ thus $PDT[5, 8]$, $PDT[6, 8]$ and $PDT[2, 8]$ are set to d. Note, s_8 defines q, hence we add 8 to the def-statement-list of q (see row 10 in Table I). Statements $s_9 - s_{11}$ are encoded in a similar way.

For the statement s_{12}, top of DS is 7, so $PDT[7, 12]$ is set to c. Since s_{12} contains }, thus $PDT[12, 7] = c$. Then we jump to s_7 because top of DS and top of LS are 7 and the first and second elements of the stack LS are not equal. Moreover the top of DS is popped (see row 14 in Table I). Now the statements $s_7 - s_{12}$ are checked again by our algorithm. Table II demonstrates the PDT of the program of Fig. 1.a.

In the following, we prove the correctness of the proposed algorithm. This proof consists of three parts: (1) completeness, (2) correctness and (3) finiteness. In completeness step, we prove that our algorithm is complete, i.e. it covers all the possible cases, Then in correctness step we prove that the algorithm is correct, and finally we show that the proposed algorithm terminates after a finite number of iterations.

Consider different kinds of the statements in C language: simple statements and compound statements. Simple statements include statements that do not contain other statements (e.g. assignments, jumps) while compound statements appear as the bodies of another statements (e.g. conditions, loops). Simple or compound statements might define or use a subset of program variables. Used variables are covered by lines 5-8 and defined variables are covered by lines 9-12. A compound statement consists of a pair of braces {}. This condition is covered by lines 2-4, 16-18, 19-20 and 29. Note, loop statements are the compound statements that must be traversed twice, once forward and once backward. This condition is covered by lines 21-28.

To prove the correctness of our algorithm, consider the statement s_i. If the stack DS is empty, thus s_i is a function header, otherwise it is clear that s_i is a function body statement. Function body statements might have braces, use or define a subset of program variables. All these conditions are covered by the proposed algorithm.

Finally, since the number of statements of a program always is finite, thus our algorithm stops after a finite number of steps.

TABLE I
THE PROCESS OF CREATING PDT

row	s_i	PDT	$L_{def}(x)$	DS	LS
1	Initialization			[]	[]
2	s_0	-	-	$[s_0]$	[]
3	s_1	$PDT[0, 1] = c$	$a = \{1\}$	$[s_0]$	[]
4	s_2	$PDT[0, 2] = c$	$a = \{1\}$ $q = \{2\}$	$[s_0]$	[]
5	s_3	$PDT[0, 3] = c$	$a = \{1\}$ $q = \{2\}$ $p = \{3\}$	$[s_0]$	[]
6	s_4	$PDT[0, 4] = c$	$a = \{1\}$ $q = \{2\}$ $p = \{3\}$ $sp = \{4\}$	$[s_0]$	[]
7	s_5	$PDT[0, 5] = c$	$a = \{1\}$ $q = \{2\}$ $p = \{3\}$ $sp = \{4\}$ $n = \{5\}$	$[s_0]$	[]
8	s_6	$PDT[0, 6] = c$	$a = \{1\}$ $q = \{2\}$ $p = \{3\}$ $sp = \{4\}$ $n = \{5, 6\}$	$[s_0]$	[]
9	s_7	$PDT[0, 7] = c$ $PDT[5, 7] = d$ $PDT[6, 7] = d$	$a = \{1\}$ $q = \{2\}$ $p = \{3\}$ $sp = \{4\}$ $n = \{5, 6\}$	$[s_0, s_7]$	$[s_7]$
10	s_8	$PDT[7, 8] = c$ $PDT[5, 8] = d$ $PDT[6, 8] = d$ $PDT[2, 8] = d$	$a = \{1\}$ $q = \{2, 8\}$ $p = \{3\}$ $sp = \{4\}$ $n = \{5, 6\}$	$[s_0, s_7]$	$[s_7]$
11	s_9	$PDT[7, 9] = c$ $PDT[5, 9] = d$ $PDT[6, 9] = d$ $PDT[2, 9] = d$ $PDT[8, 9] = d$	$a = \{1\}$ $q = \{2, 8\}$ $p = \{3, 9\}$ $sp = \{4\}$ $n = \{5, 6\}$	$[s_0, s_7]$	$[s_7]$
12	s_{10}	$PDT[7, 10] = c$ $PDT[3, 10] = d$ $PDT[9, 10] = d$ $PDT[4, 10] = d$	$a = \{1\}$ $q = \{2, 8\}$ $p = \{3, 9\}$ $sp = \{4, 10\}$ $n = \{5, 6\}$	$[s_0, s_7]$	$[s_7]$
13	s_{11}	$PDT[7, 11] = c$ $PDT[5, 11] = d$ $PDT[6, 11] = d$	$a = \{1\}$ $q = \{2, 8\}$ $p = \{3, 9\}$ $sp = \{4, 10\}$ $n = \{5, 6, 11\}$	$[s_0, s_7]$	$[s_7]$
14	s_{12}	$PDT[7, 12] = c$ $PDT[12, 7] = c$	$a = \{1\}$ $q = \{2, 8\}$ $p = \{3, 9\}$ $sp = \{4, 10\}$ $n = \{5, 6, 11\}$	$[s_0]$	$[s_7]$
15	s_7	$PDT[0, 7] = c$ $PDT[5, 7] = d$ $PDT[6, 7] = d$ $PDT[11, 7] = d$	$a = \{1\}$ $q = \{2, 8\}$ $p = \{3, 9\}$ $sp = \{4, 10\}$ $n = \{5, 6, 11\}$	$[s_0, s_7]$	$[s_7, s_7]$
16	s_8	$PDT[7, 8] = c$ $PDT[5, 8] = d$ $PDT[6, 8] = d$ $PDT[11, 8] = d$ $PDT[2, 8] = d$ $PDT[8, 8] = d$	$a = \{1\}$ $q = \{2, 8\}$ $p = \{3, 9\}$ $sp = \{4, 10\}$ $n = \{5, 6, 11\}$	$[s_0, s_7]$	$[s_7, s_7]$
17	s_9	$PDT[7, 9] = c$ $PDT[5, 9] = d$ $PDT[6, 9] = d$ $PDT[11, 9] = d$ $PDT[2, 9] = d$ $PDT[8, 9] = d$	$a = \{1\}$ $q = \{2, 8\}$ $p = \{3, 9\}$ $sp = \{4, 10\}$ $n = \{5, 6, 11\}$	$[s_0, s_7]$	$[s_7, s_7]$

IV. PROGRAM SLICING BASED ON PDT

In this section, we present two algorithm for backward and forward program slicing based on PDT. Since chop, backbone and union slices can be generated from backward and forward slices, we do not consider any algorithm for them.

A. Backward slicing

Out backward program slicing is presented in Algorithm 2. Let p be a source code and PDT_p be its corresponding

18	s_{10}	$PDT[7,10] = c$ $PDT[4,10] = d$ $PDT[10,10] = d$ $PDT[3,10] = d$ $PDT[9,10] = d$	$a = \{1\}$ $q = \{2,8\}$ $p = \{3,9\}$ $sp = \{4,10\}$ $n = \{5,6,11\}$	$[s_0, s_7]$	$[s_7, s_7]$
19	s_{11}	$PDT[7,11] = c$ $PDT[5,11] = d$ $PDT[6,11] = d$ $PDT[11,11] = d$	$a = \{1\}$ $q = \{2,8\}$ $p = \{3,9\}$ $sp = \{4,10\}$ $n = \{5,6,11\}$	$[s_0, s_7]$	$[s_7, s_7]$
20	s_{12}	$PDT[7,12] = c$ $PDT[12,7] = c$	$a = \{1\}$ $q = \{2,8\}$ $p = \{3,9\}$ $sp = \{4,10\}$ $n = \{5,6,11\}$	$[s_0]$	[]
21	s_{13}	$PDT[0,13] = c$ $PDT[2,13] = d$ $PDT[8,13] = d$	$a = \{1\}$ $q = \{2,8\}$ $p = \{3,9\}$ $sp = \{4,10\}$ $n = \{5,6,11\}$	$[s_0]$	[]
22	s_{14}	$PDT[0,14] = c$ $PDT[4,14] = d$ $PDT[10,14] = d$	$a = \{1\}$ $q = \{2,8\}$ $p = \{3,9\}$ $sp = \{4,10\}$ $n = \{5,6,11\}$	$[s_0]$	[]
23	s_{15}	$PDT[0,15] = c$ $PDT[15,0] = c$	$a = \{1\}$ $q = \{2,8\}$ $p = \{3,9\}$ $sp = \{4,10\}$ $n = \{5,6,11\}$	[]	[]

TABLE II
PDT OF FIG. 1.A

-	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		c	c	c	c	c	c	c						c	c	c
1																
2									d	d				d		
3											d					
4											d				d	
5								d	d			d				
6								d	d			d				
7									c	c	c	c	c			
8									d	d				d		
9											d					
10											d				d	
11								d	d			d				
12								c								
13																
14																
15	c							c								

PDT. This algorithm takes the program statement $s_i \in p$, the variable $x \in s_i$ and PDT_p as the input parameters and returns the backward slice $\langle s_i, x\rangle$. We use two initially empty lists $Slicing\_set$ and $Temp$ to keep the final and temporary result of program slicing. Since the statement s_i is the starting point of the slicing, so we add its number (i.e. i) to the list $Temp$. Then, the algorithm repeats the following steps until $Temp$ becomes empty: (a) it removes an element from the temporary list (such as $j \in Temp$) and adds it to the list $Slicing\_set$; (b) Next, the algorithm checks the column j of PDT_p. It adds the row number k to $Temp$ if $PDT_p[k, j]$ is c or d and k is not in the lists $Temp$ and $Slicing\_set$, which means that k has not been processed.

For more clear, Table III explains our algorithm for the

Algorithm 2 Backward program slicing

```
1: Temp ← Temp ∪ i
2: while not(empty(Temp)) do
3:    j ← remove(Temp)
4:    Slicing_set ← Slicing_set ∪ j
5:    rows ← rows number of PDT_p
6:    for all k ∈ rows do
7:       if (PDT_p[j,k] = c or PDT_p[j,k] = d) then
8:          if (k ∉ Temp and k ∉ Slicing_set) then
9:             Temp ← Temp ∪ k
10:         end if
11:      end if
12:   end for
13: end while
14: return Slicing_set
```

backward slice $\langle s_{13}, q\rangle$. In row 2, the statement s_{13} has dependency with the statements s_0, s_2 and s_8, so we add them to the temporary list since they are not in the lists $Temp$ or $Slicing\_set$. While in row 4, s_{11} has dependency with the statements s_5, s_6, s_7 and s_{11}, but none of them are added to the list $Temp$ because $s_5, s_6, s_7 \in Temp$ and $s_{11} \in Slicing\_set$.

TABLE III
THE PROCESS OF BACKWARD SLICING FOR s_{13}

row	j	$Temp$	$Slicing\_set$
1		$\{13\}$	$\{\}$
2	13	$\{0, 2, 8\}$	$\{13\}$
3	8	$\{0, 2, 5, 6, 7, 11\}$	$\{13, 8\}$
4	11	$\{0, 2, 5, 6, 7\}$	$\{13, 8, 11\}$
5	7	$\{0, 2, 5, 6, 12\}$	$\{13, 8, 11, 7\}$
6	12	$\{0, 2, 5, 6\}$	$\{13, 8, 11, 7, 12\}$
7	6	$\{0, 2, 5\}$	$\{13, 8, 11, 7, 12, 6\}$
8	5	$\{0, 2\}$	$\{13, 8, 11, 7, 12, 6, 5\}$
9	2	$\{0\}$	$\{13, 8, 11, 7, 12, 6, 5, 2\}$
10	0	$\{15\}$	$\{13, 8, 11, 7, 12, 6, 5, 2, 0\}$
11	15	$\{\}$	$\{13, 8, 11, 7, 12, 6, 5, 2, 0, 15\}$

In the following we prove that the proposed backward slicing algorithm is complete, correct and terminates after a finites number of iterations. To prove the completeness, consider L as the set of dependency types in the PDT, so $L = \{c, d\}$. Initially, the intended slice consists of only the slicing creation statement s_i. There can be two possibilities: s_i may be a header function or may be a statement of function body. If s_i is a function header, then the slice contains this statement and its close brace, otherwise it must be connected to the other statements through the members of L. This is true in the proposed algorithm because we proved that PDT covers all possible types of dependencies including data dependency and control dependency.

To prove the correctness of our algorithm, we start with the slicing criterion statement s_i in a source code p. Initially, the slice list $Slicing\_set$ is empty. Clearly, the slicing criterion statement must be in the output slice, hence the backward slicing algorithm add it to the list $Slicing\_set$. Next, we have to check all effective statements by following data and control

dependencies. These relations are declared by c or d in PDT_p. Hence, our algorithm adds the connected statements to the slice list by finding the data and control dependencies.

Finally, a backward slice is created in a finite number of steps, since we do not add duplicate statements to the temporary list and the number of statements is finite.

B. Forward slicing

Algorithm 3 presents our forward program slicing. The definitions of PTD_p, $Slicing\_set$ and $Temp$ are same as the definitions of them in the previous sub-section. To generate the forward slice of s_i, we add i to the temporary list, then we repeat the following steps while $Temp$ is not empty: (a) we remove an element such as j from $Temp$ and adds it to the list $Slicing\_set$; (b) Next, the row j of PDT_p is checked. For each column k, if $PDT_p[j,k]$ is c or d and k is not in the lists $Temp$ and $Slicing\_set$, then we add it to the temporary list. Table IV demonstrates the proposed algorithm for the forward slice of s_8.

Algorithm 3 Forward program slicing

```
1: Temp ← Temp ∪ i
2: while not(empty(Temp)) do
3:    j ← remove(Temp)
4:    Slicing_set ← Slicing_set ∪ j
5:    cols ← columns number of PDT_p
6:    for all k ∈ cols do
7:       if (PDT_p[j,k] = c or PDT_p[j,k] = d) then
8:          if (k ∉ Temp and k ∉ Slicing_set) then
9:             Temp ← Temp ∪ k
10:         end if
11:      end if
12:   end for
13: end while
14: return Slicing_set
```

TABLE IV
THE PROCESS OF FORWARD SLICING FOR s_8

row	j	$Temp$	$Slicing\_set$
1		$\{8\}$	$\{\}$
2	8	$\{9,13\}$	$\{8\}$
3	13	$\{9\}$	$\{8,13\}$
4	9	$\{10\}$	$\{8,13,9\}$
5	10	$\{14\}$	$\{8,13,9,10\}$
6	14	$\{\}$	$\{8,13,9,10,14\}$

V. CONCLUSION

Program slicing is one of the program analysis techniques that is used to save time and cost in software testing process. It restricts the focus of a task to specific sub-components of a program by extracting the statements. Different type of dependence graphs are used to represent the relations between the statements of a program. However, drawing these graphs for large and complex programs is very complicated and it is impossible in non-automatic approaches. Hence, in this paper we proposed Program Dependence Table (PDT) as an intermediate representation tool to handle the data and control dependency in the intra-procedural programs. PDT is generated automatically form a source code without drawing the program dependence graph. Then, we presented two automatic slicing algorithms including forward and backward slicing based on PDT which are useful for automatic unit testing. Moreover, we show that chop, backbone and union slicing can be generated by forward and backward slicing. The proposed backward program slicing produces an executable slice because it covers pair braces in a program. For future work, it is suggested to extend the algorithm for slicing in inter-procedural, object-oriented and aspect-oriented programming paradigms.

REFERENCES

[1] P. Ammann and J. Offutt, *Introduction to software testing, Second Edition*. Cambridge University Press, 2016.

[2] W. Lewis, *Software Testing and Continuous Quality Improvement, Third Edition*. CRC Press, 2016.

[3] E. Dustin, T. Garrett, and B. Gauf, *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Pearson Education, 2009.

[4] A. Mohammadian and B. Arasteh, "Using program slicing technique to reduce the cost of software testing," *Journal of Artificial Intelligence in Electrical Engineering*, vol. 2, no. 7, pp. 24–33, 2013.

[5] N. Sasirekha, A. Edwin Robert, and M. Hemalatha, "Program slicing techniques and its application," *International Journal of Software Engineering & Application*, vol. 2, no. 3, pp. 50–64, 2011.

[6] F. Angerer, H. Prähofer, and P. Grünbacher, "Modular change impact analysis for configurable software," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, 2016, pp. 468–472.

[7] L. Du and P. Cai, "A survey on applications of program slicing," in *Soft Computing in Information Communication Technology*, J. Luo, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 215–220.

[8] Y. Yang, Y. Zhao, C. Liu, H. Lu, Y. Zhou, and B. Xu, "An empirical investigation into the effect of slice types on slice-based cohesion metrics," *Information & Software Technology*, vol. 75, pp. 90–104, 2016.

[9] Y. N. Srikant and P. Shankar, Eds., *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, 2007.

[10] M. yang hong and L. ruo qin, "Application of dynamic program slicing technique in test data generation," *Procedia Computer Science*, vol. 111, pp. 355–360, 2017.

[11] F. Rabbi, H. Wang, W. MacCaull, and A. Rutle, "A model slicing method for workflow verification," *Electr. Notes Theor. Comput. Sci.*, vol. 295, pp. 79–93, 2013.

[12] P. Jorgensen, *Software Testing: A Craftsmans Approach, Fourth Edition*, ser. An Auerbach book. Taylor & Francis, 2013.

[13] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.*, 1981, pp. 439–449.

[14] ——, "Program slicing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, 1984.

[15] X. Qi and Z. Jiang, "Precise slicing of interprocedural concurrent programs," *Frontiers Comput. Sci.*, vol. 11, no. 6, pp. 971–986, 2017.

[16] D. P. Mohapatra, R. Mall, and R. Kumar, "An overview of slicing techniques for object-oriented programs," *Informatica (Slovenia)*, vol. 30, no. 2, pp. 253–277, 2006.

[17] D. Munjal, J. Singh, S. Panda, and D. P. Mohapatra, "Automated slicing of aspect-oriented programs using bytecode analysis," in *39th IEEE Annual Computer Software and Applications Conference, COMPSAC 2015, Taichung, Taiwan, July 1-5, 2015. Volume 2*, 2015, pp. 191–199.

[18] M. Sahu and D. P. Mohapatra, "Computing dynamic slices of feature-oriented programs using execution trace file," *ACM SIGSOFT Software Engineering Notes*, vol. 42, no. 2, pp. 1–16, 2017.

[19] Y. Zhang, "A precise monadic dynamic slicing method," *Knowl.-Based Syst.*, vol. 115, pp. 40–54, 2017.

[20] S. Yoo, D. W. Binkley, and R. D. Eastman, "Observational slicing based on visual semantics," *Journal of Systems and Software*, vol. 129, pp. 60–78, 2017.
[21] S. Horwitz, T. W. Reps, and D. W. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, 1990.
[22] J. Singh and D. P. Mohapatra, "Dynamic slicing of concurrent aspectj programs: An explicit context-sensitive approach," *Softw., Pract. Exper.*, vol. 48, no. 1, pp. 233–260, 2018.
[23] I. Mastroeni and D. Zanardini, "Abstract program slicing: An abstract interpretation-based approach to program slicing," *ACM Trans. Comput. Log.*, vol. 18, no. 1, pp. 7:1–7:58, 2017.
[24] A. Zeller, *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009.
[25] A. Afshar, *Application Software Re-Engineering*. Pearson Education, 2010.