Dynamic Slicing using Dynamic System Dependence Graph

Soon-Hyung Park[†] and Man-Gon Park[††]

ABSTRACT

Traditional slicing techniques make slices through dependence graph and improve the accuracy of slices. However, traditional slicing techniques require many vertices and edges in order to express a data communication link because they are based on static slicing techniques. Therefore the graph becomes very complicated.

We propose the representation of a *dynamic system dependence graph* so as to process the slicing of a software system that is composed of related programs in order to process certain jobs. We also propose programs on efficient slicing algorithm using relations of relative tables in order to compute dynamic slices of a software system.

Using a *marking table* from results of the proposed algorithm can make *dynamic system dependence graph* for dynamic slice generation. Tracing this graph can generate final slices. We have illustrated our example with C program environment.

Consequently, the efficiency of the proposed *dynamic system dependence graph* technique is also compared with the dependence graph techniques discussed previously. As the results, this is certifying that the *dynamic system dependence graph* is more efficient in comparison with *system dependence graph*.

동적 시스템 종속 그래프를 사용한 동적 슬라이싱

박순형[†] · 박만곤[†]

요 약

기존의 슬라이싱 기법들은 슬라이스를 생성하며 생성된 슬라이스의 정확성을 위해 종속 그래프를 사용한다. 그러나, 기존의 많은 슬라이싱 기법들은 동적 슬라이싱 기법에 바탕을 두고 데이터 통신 링크를 표현하기 때문에 많은 정점들(vertices)과 간선들(edges)을 필요로 한다. 그러므로 그 그래프는 매우 복잡하다.

어떤 프로그램 시스템에 대한 소프트웨어 슬라이싱을 처리하기 위해 본 논문에서는 동적시스템종속그래프를 제안한다. 그리고, 소프트웨어 시스템의 동적 슬라이스를 산출하기 위해 관련 테이블들의 관계도를 이용한 효율적인 슬라이싱 알고리즘을 제안한다.

동적 슬라이스의 생성을 위한 동적시스템종속그래프는 제안된 알고리즘으로부터 얻어진 마킹테이블을 사용해서 얻어진다. 슬라이스의 최종 결과는 이 그래프를 추적함으로써 얻어진다. 결론적으로 제안된 동적시스템종속그래프 기법의 효율성을 기존의 종속그래프 기법과 비교하였다.

Key words: program slicing, dynamic program slicing, program dependence graph, system dependence graph, dynamic system dependence graph

1. Introduction

A program slice consists of the parts of a program that potentially affect the values computed at some point of interest referred to as a *slicing criterion*. The task of computing program slices is called *program slicing*. Namely, *program slicing* is a method of finding all statements in program P

[†] 부경대학교 공과대학 전자컴퓨터정보통신공학부

that may directly or indirectly affect the value of a variable *var* at some point *p* [1-4]. A *system slicing* is a slicing of multi-procedure programs. The term *system* will be used to emphasize a program with multiple procedures.

A slicing is divided into a *static slicing* and a *dynamic slicing* according to the existence of actual execution by input values. The static slice proposed by Weiser is the set of all statements that might affect the value of a given variable occurrence. In this paper, we investigate the concept of the dynamic slice consisting of all statements that actually affect the value of a variable occurrence for a given program input[5-7].

Traditional slicing techniques used dependence graphs in order to compute slices accurately. However, traditional dependence graphs, especially a *system dependence graph* is very complicated because it requires a lot of vertices and edges in order to indicate data communication between two procedures. Therefore, it is difficult for the programmer and the tester to apply these techniques. More efficient dependence graph techniques were in focus in previous studies. If the number of source programs is one, the graph which computes static slices is called a *program dependence graph*; if the number of source programs is more than one, the graph which computes static slices is called a *system dependence graph*[8-11].

In this paper, we proposed the notation that represents a *dynamic system dependence graph* and the efficient slicing algorithm using the relation of related tables to compute dynamic slices in a software system. We can make a *dynamic system dependence graph* to generate dynamic slices using the *marking table* that is generated as result of the execution of this algorithm.

In section 2, we review the dependence graphs concerning traditional slicing approaches. In section 3, we introduce the algorithm proposed for dynamic system slicing and the *dynamic system dependence graph*. In section 4, we describe and illustrate about the dynamic system dependence algorithm. The slicing technique is also presented and compared with traditional method, in section 5.

2. Dependence Graph

We use dependence graph notation traditionally in order to confirm the generation of correct program slices. Therefore, many types of dependence graph notation are introduced according to advanced program slicing techniques. In this section, we explain dependence graphs so as to express the traditional program slicing.

The static slice proposed by Weiser is the set of all statements that might affect the value of a given variable occurrence. It can be computed using the *program dependence graph* technique or the *system dependence graph* technique. The dynamic slices consist of all statements that actually affect the value of a variable occurrence for a given program input. Thus, *dynamic dependence graph* technique can produce slices.

2.1 Program Dependence Graph Technique

A *program dependence graph* technique proposed by Susan Horwitz, Tomas Reps and David Binkley is the method to compute slices through data flow analysis and control flow analysis. It has two types of directed edges, namely data-dependence edges and control- dependence edges. A data-dependence edge from vertex v_i to vertex v_j implies that the computation performed at vertex v_i directly depends on the value computed at vertex v_j. More precisely, it means that the computation at vertex v_i uses a variable, *var*, that is defined at vertex v_j, and there is an execution path from v_j to v_i along which *var* is never redefined. A control-dependence edge from v_i to v_j means that the node v_i may or may not be executed depending on the boolean outcome of the predicate expression at node v_j [8-9].

2.2 System Dependence Graph Technique

A *system dependence graph* contains the *program dependence graph* that expresses the main program of a system, the procedure dependence graph, which express a collection of auxiliary procedures of system, and the some additional edges. Additional edges can be classified as follows: (1) edges that express direct dependence passing between call statement and called procedure and (2) transitive flow dependences due to calls.

In addition, there are five new kinds of vertices being used in a *program dependence graph*. A call statement is represented using a *call* vertex. Parameter passing is represented using four kinds of *parameter* vertices. On the calling side, parameter passing is represented by *actual-in* and *actual-out* vertices, which are control dependent on the call vertex. In the called procedure, parameter passing is represented by *formal-in* and *formal-out* vertices, which are control dependent on the procedure's entry vertex[12-13]. But the graph is further complicated because the number of vertices increase in order to add vertices as much as two times and because the edges between them is connected.

2.3 Dynamic Dependence Graph Technique

A *dynamic dependence graph* represents nodes with dependence edges based on the criterion node after the generation execution history for input value given. A new node for every occurrence of a statement in the execution history may need to be created if another node has the same transitive dependencies[14].

A slicing criterion of program P executed on input x is a triple $C = (x, I^q, V)$, where I is an instruction at position q on H and V is a subset of variables in P. When executed on program input x, we produce an execution history H', for which there exists the corresponding execution position q' such that the value of v of I^q in H_x equals the value of v of $I^{q'}$ in H'_x[5].

We denote the execution history of the program under the given test-case by the sequence $<v_1, v_2, \ldots\ldots, v_n>$ of vertices in the *program dependence graph* appended in the order in which they are visited during execution. Node Y at position p in H_x(ie., $H_x(p) = Y$) will be written as Y^p. We use superscripts to distinguish between multiple occurrences of the same node in the execution history [3,7].

3. Dynamic System Slicing Technique

A *dynamic system dependence graph* that we propose is a technique that it is possible to represent concepts of system dependence. A concept of *dynamic system dependence graph* is based on that of dynamic slicing technique.

A traditional *system dependence graph* technique is complicated when it represents multi-procedure programs. So we propose an efficient *dynamic system dependence graph*. A size of complexities of *dynamic system dependence graph* is smaller in comparison with *system dependence graph*. A final aim of program slicing is to reduce size of slices or dependence graph.

The procedure that computes dynamic slices using the proposed *dynamic system dependence graph* is composed of five steps. These are:

(1) *system node analysis*
(2) *system execution history analysis*
(3) *dynamic system dependence algorithm application*
(4) *dynamic system dependence graph generation*
(5) *sliced program generation*

3.1 System Node Analysis Step

The *system node analysis step* draws up the *table related node* for the source program. A *table related node* is a set of the components of nodes. It is composed of *node number, node type, DEF,*

REF and *affiliation module number*(*A.M.N*).

(1) Node Number
Sequential number of nodes that compose of a system

(2) Node Type
The nodes that are composed of system are classified by eight kinds; procedure name, input, print, assign, repeat, select, call, return.

(3) DEF
The set of variables whose values are defined.

(4) REF
The set of variables whose values are used.

(5) Affiliation Module Number
Number of procedure name node among nodes in affiliated procedure.

3.2 System Execution History Analysis Step

The *system execution history analysis step* draws up the *execution history table* and the *path table related module* after analysis of the execution history.

(1) *Execution History Table*

The *execution history table* is a set of nodes which are visited during execution and is composed of *node execution number*(*N.E.N*), *node number*, and *affiliation module execution order* (*A.M.E.O*). The *node execution number* represents execution history order, and *node number* is equal to number of nodes in *node table related*. The *affiliation module execution order* means an order of the node that contains procedure name in *execution history table*.

(2) *Path Table Related Module*

A path table related module is a set of the movement paths for variables to be connected within the inter-procedures that occurred during program execution. It is composed of an *affiliation module execution order*(*A.M.E.O*), an *affiliation module number*(*A.M.N*), and some *path orders*. The *path order* contains a *variable name*(*V.N*) and a *variable affiliation module number*(*V.A.M.N*). The *affiliation module execution order* and the *affiliation module number* mean the execution order in execution history according to *procedure call* in *execution history table* and the node number which contains affiliated procedure name. The *variable names* are stored as the linked datum about parameter variables. The *variable affiliation module number* is node number in *related node table* that contains related *variable name*.

3.3 Dynamic System Dependence Algorithm Application Step

The *dynamic system dependence algorithm application step* is a step that produces a *marking table* to extract dynamic slices. A *marking table* is composed of *node execution orders*(*N.E.O*), *node numbers*, *markings*. The *marking algorithm* proposed below and Fig. 1 denotes the relationship diagram of table used in algorithm.

```
procedure Main()
  read SlicingBaseVar,
       SlicingBaseVarNodeExeOrd
  icnt = BaseVarNodeExeOrd
  NodeExeOrd.ExeHistTbl = icnt
  while NodeExeOrd.ExeHistTbl not = " " do
  BaseVar.BaseVarTbl = SlicingBaseVar
    NodeNum.NodeRelatedTbl = NodeNum.ExeHistTbl
    while NodeNum.NodeRelatedTbl not = " " do
      BaseAfanModNum.BaseVarTbl =
      AfanModNum.NodeRelatedTbl
end while
    write BaseVar.BaseVarTbl,
          BaseAfanModNum.BaseVarTbl
end while
  call FindFirstDef()
  call FindSlice()
end.

procedure FindFirstDef()
  NodeNum.NodeRelatedTbl =
  NodeNum.ExeHistTbl
  while NodeNum.NodeRelatedTbl not = " " do
    if NodeType.NodeRelatedTbl = 'assign'
    then call AssignProc()
```

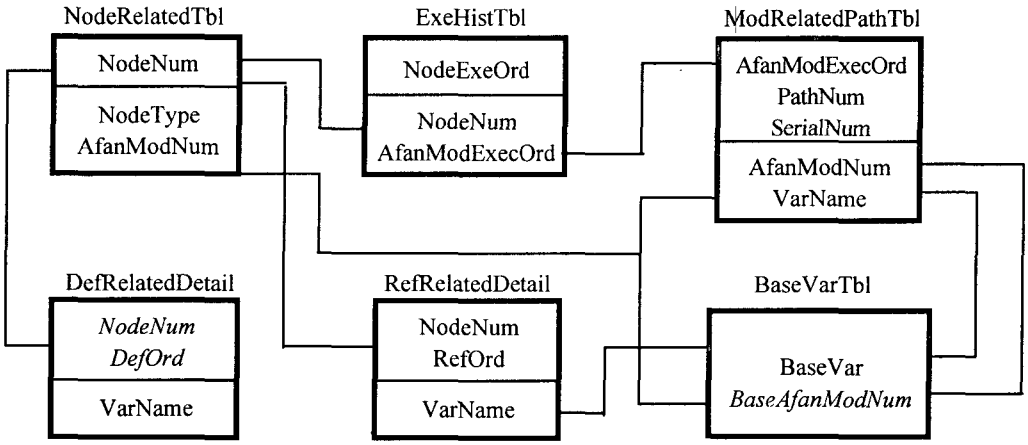


Fig. 1. Table relationship diagram

```
  end if
    end while
  end.

  procedure FindSlice()
    while icnt > 1 do
      icnt = icnt - 1
      NodeNum.NodeRelatedTbl =
      NodeNum.ExeHistTbl
      while NodeNum.NodeRelatedTbl not = " " do
  case NodeType.NodeRelatedTbl
          : 'procedure name'
              then call ProcedureProc()
          : 'call' then call CallProc()
          : 'read' then call ReadProc()
          : 'write' then call WriteProc()
          : 'assign' then call AssignProc()
          : 'repeat' then call ControlProc()
          : 'selection' then call ControlProc()
        end case
      end while
    end while
  end.

  procedure ProcedureProc()
    AfanModExecOrd.ModRelatedPathTbl =
    AfanModExecOrd.ExeHistTbl
    AfanModNum.ModRelatedPathTbl =
    AfanModNum.NodeRelatedTbl
    while PathOrd.ModRelatedPathTbl not = " " do
      VarName.ModRelatedPathTbl=BaseVar.BaseVarTbl
      AfanModNum.ModRelatedPathTbl =
      BaseAfanModNum.BaseVarTbl
      while SerialNum.ModRelatedPathTbl not = " " do
        BaseVar.BaseVarTbl =
        BaseAfanModNum.BaseVarTbl = " "
        SerialNum = SerialNum + 1
        while SerialNum.ModRelatedPathTbl not = " " do
          BaseVar.BaseVarTbl = VarName.ModRelatedPathTbl
          BaseAfanModNum.BaseVarTbl =
          AfanModNum.ModRelatedPathTbl
          write BaseVar.BaseVarTbl ,
                BaseAfanModNum.BaseVarTbl
        end while
      end while
    end while
  end while
  end.

  procedure CallProc()
  end.

  procedure ReadProc()
    AfanModExecOrd.ModRelatedPathTbl =
    AfanModExecOrd.ExeHistTbl
    AfanModNum.ModRelatedPathTbl =
    AfanModNum.NodeRelatedTbl
    while PathOrd.ModRelatedPathTbl not = " " do
      while NodeNum.DefRelatedDetail not = " " do
        VarName.ModRelatedPathTbl =
        VarName.DefRelatedDetail
        AfanModNum.ModRelatedPathTbl =
        AfanModNum.NodeRelatedTbl
        while SerialNum.ModRelatedPathTbl not = " " do
          BaseVar.BaseVarTbl =
          BaseAfanModNum.BaseVarTbl =
        end while
      end while
  end while
  end.

  procedure WriteProc()
  end.

  procedure AssignProc()
    AfanModExecOrd.ModRelatedPathTbl =
```

```
  AfanModExecOrd.ExeHistTbl
  AfanModNum.ModRelatedPathTbl =
  AfanModNum.NodeRelatedTbl
  while PathOrd.ModRelatedPathTbl not = " " do
    while NodeNum.DefRelatedDetail not = " " do
      VarName.ModRelatedPathTbl =
      VarName.DefRelatedDetail
      AfanModNum.ModRelatedPathTbl =
      AfanModNum.NodeRelatedTbl
      while SerialNum.ModRelatedPathTbl not = " " do
        BaseVar.BaseVarTbl = " "
        BaseAfanModNum.BaseVarTbl = " "
        while NodeNum.RefRelatedDetail not = " " do
          BaseVar.BaseVarTbl =
          VarName.RefRelatedDetail
          BaseAfanModNum.BaseVarTbl =
          AfanModNum.NodeRelatedTbl
          write BaseVar.BaseVarTbl ,
                BaseAfanModNum.BaseVarTbl
        end while
      end while
    end while
end while
end.

procedure ControlProc()
  BaseVar.BaseVarTbl =
  VarName.RefRelatedDetail
  BaseAfanModNum.BaseVarTbl =
  AfanModNum.NodeRelatedTbl
  write BaseVar.BaseVarTbl ,
        BaseAfanModNum.BaseVarTbl
end.
```

3.4 Dynamic System Dependence Graph Generation Step

The *dynamic system dependence graph generation step* is a step that generates *dynamic system dependence graph* based on the result of the *marking table* that is generated by the application of dynamic system slice marking algorithm.

A dynamic system dependence graph is composed of vertices and edges. A node in solid represents the initial vertex. A node in bold represents the vertex sliced. An initial edge is represented by a dashed edge. There are three types of edges that connect to a sliced node. An intra dependence edge is shown by a solid line, inter dependence edge is shown by a bold line, and the return dependence edge is shown by a bold dash.

3.5 Sliced Program Generation Step

The sliced program generation step is a step that produces final sliced program after program formalizing based on dynamic system dependence graph. The sliced program is another program whose behavior is identical, for the same program input, to that of the original program with respect to a variable of interest.

4. Application of Dynamic System Slicing Technique

4.1 Application example

We apply the dynamic system slicing algorithm proposal to an example program in Fig. 2.

```
 1    program Main
 2      i=1
 3      x=1
 4      y=1
 5      while i<=2 do
 6        call A(x,y,z,i)
        loop
 7      print z
 8    end

 9    procedure A(a,b,c,d)
10      input k
11      a=k**2
12      call IFF(a,b,c)
13      call Increment(d)
14    end

15    procedure  IFF(p,q,r)
16      if p<0 then
17        q=f1(p)
18        r=f2(q)
        else
19        q=f3(p)
20        r=f4(q)
        end if
21    end

22    procedure Increment(e)
23      e=e+1
24    end
```

Fig. 2. Example Program

(1) *System Node Analysis Step*

The analysis data table of nodes that consist of the sample program from Fig. 2 is noted in Table 1.

(2) *System Execution History Analysis Step*

The execution history of example program thus far is {1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 9^1, 10^1, 11^1, 12^1, 15^1, 16^1, 19^1, 20^1, 21^1, 13^1, 14^1, 22^1, 23^1, 24^1, 5^2, 6^2, 9^2, 10^2, 11^2, 12^2, 15^2, 16^2, 19^2, 20^2, 21^2, 13^2, 14^2, 22^2, 23^2, 24^2, 5^3, 7^1} where a = {-2, 3}. The *execution history table* and the *path table related module* to compute the dynamic slices when a system slicing criterion is Z of execution order are illustrated in Table 2 and Table 3.

Table 1. The data of related nodes for the Example Program

Node number	Node type	DEF	REF	A.M.N
1	procedure name	Main		1
2	assign	i		1
3	assign	x		1
4	assign	y		1
5	repeat		i	1
6	call	A	x,y,z,i	1
7	print	z		1
8	return			1
9	procedure name	A	a,b,c,d	9
10	input	k		9
11	assign	a	k	9
12	call	IFF	a,b,c	9
13	call	Increment	d	9
14	return			9
15	procedure name	IFF	p,q,r	14
16	selection		p	14
17	assign	q	p	14
18	assign	r	q	14
19	assign	q	p	14
20	assign	r	q	14
21	return			14
22	procedure name	Increment	e	21
23	assign	e	e	21
24	return			21

Table 2. Execution history table

N.E.N	Node number	A.M.E.O
1	1	1
2	2	1
3	3	1
4	4	1
5	5	1
6	6	1
7	9	7
8	10	7
9	11	7
10	12	7
11	15	11
12	16	11
13	19	11
14	20	11
15	21	11
16	13	7
17	14	7
18	22	18
19	23	18
20	24	18
21	5	1
22	6	1
23	9	23
24	10	23
25	11	23
26	12	23
27	15	27
28	16	27
29	19	27
30	20	27
31	21	27
32	13	23
33	14	23
34	22	34
35	23	34
36	24	34
37	7	1
38	7	1

(3) *Dynamic System Slicing Algorithm Application Step*

The marking table that is made after the application of the *system node analysis table*, the *execution history table* and the *path table related module* which were made during steps (1) and (2) to the *dynamic system slicing algorithm* is noted in Table 4. The nodes that contain v in the *marking table* are execution history nodes related to slicing criterion.

Table 3. Path table related module

A.M. E.O	A.M. N	Path order (1)		Path order (2)		Path order (3)	
		V.N	V.A. M.N	V.N	V.P. M.N	V.N	V.A. M.N
7	9	a	9	x	1		
		b	9	y	1		
		c	9	z	1		
		d	9	i	1		
11	15	p	15	a	9	x	1
		q	15	b	9	y	1
		r	15	c	9	z	1
18	22	e	22	d	9	i	1
23	9	a	9	x	1		
		b	9	y	1		
		c	9	z	1		
		d	9	i	1		
27	15	p	15	a	9	x	1
		q	15	b	9	y	1
		r	15	c	9	z	1
34	22	e	22	d	9	i	1

(4) *Dynamic System Dependence Graph Generation Step*

The *dynamic system dependence graph* for generation of slice nodes that is made based upon the making node containing "√" at *marking table* is illustrated in Fig. 3.

(5) *Sliced Program Generation Step*

The *sliced program generation step* is a step which procedures the sliced programs based on the *dynamic system dependence graph* of Figure 3. It produces the final programs after formalizing the programs. The slices {1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 19, 20, 21, 22, 23, 24} can be constructed by traversing the *dynamic system dependence graph* when slicing criterion is Z of *execution history order* 30,namely, *node number* 7. The final slice nodes are {1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 19, 20, 21, 22, 23, 24} because *node number* 16 can be omitted during formalizing step.

4.2 Comparison with the graphs

The complexities ϕ of the traditional *system dependence graph*(SDG) techniques and the *dynamic system dependence graph*(DSDG) technique proposed in this paper are represented below when the number of system nodes is equal to *d*, the number of call nodes equal to *c*, the number of call

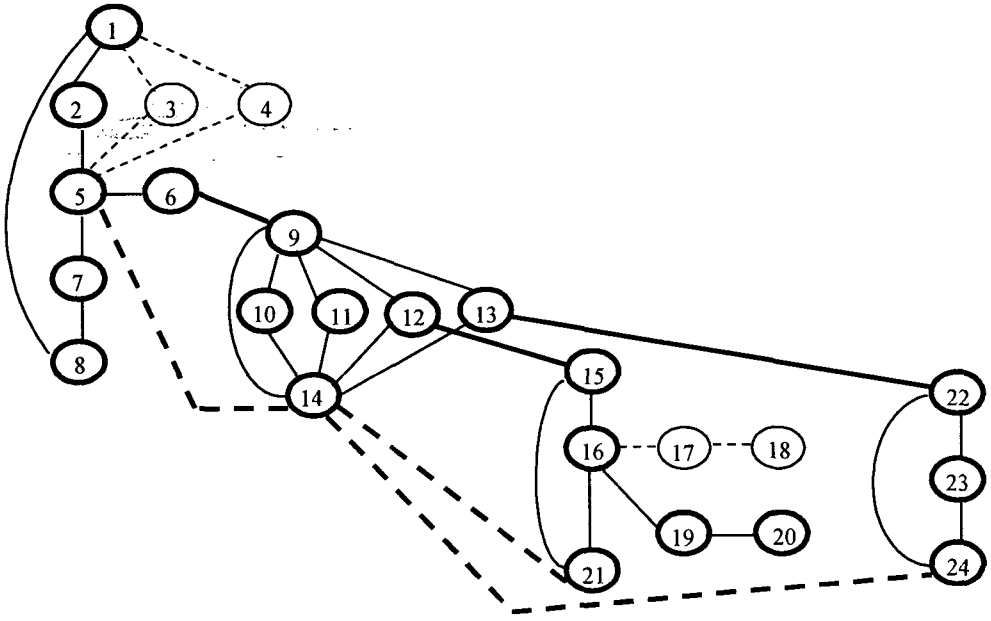


Fig. 3. DSDG of Example Program

Table 4. Marking table for sliced nodes

N.E.O	Node number	Marking
1	1	√
2	2	√
3	3	
4	4	
5	5	√
6	6	√
7	9	√
8	10	
9	11	√
10	12	√
11	15	√
12	16	√
13	19	√
14	20	√
15	21	√
16	13	√
17	14	√
18	22	√
19	23	√
20	24	√
21	5	√
22	6	√
23	9	√
24	10	√
25	11	√
26	12	√
27	15	√
28	16	√
29	19	√
30	20	√
31	21	
32	13	
33	14	
34	22	
35	23	
36	24	
37	5	
38	7	

parameters equal to p, the name parameter variable equals to n, the number of vertices equal to σ and the number of edges equal to τ.

- $\phi(SDG)$
  $= \{\sum \sigma(SDG)\} + \{\sum \tau(SDG)\}$
  $= \{\lceil(d+2(p+n))\rceil\} + \{\lceil(d+4p+2n+c)\rceil\}$
  $= \lceil(2d+6p+4n+c)\rceil$
- $\phi(DSDG)$
  $= \{\sum \sigma(DSDG)\} + \{\sum \tau(DSDG)\}$
  $= \{d\} + \{c+c+\lceil(2(d-2)+1)\rceil\}$
  $= \lceil(3d+2c-3)\rceil$

The complexities ϕ of example program in Fig. 2 are compared as shown below.

Program order	d	c	p	n
1	8	1	4	0
2	6	2	4	4
3	7	0	0	3
4	3	0	0	1

- $\phi(SDG) = \lceil(48+48+32+3)\rceil = 131$
  (We calculate $\phi(SDG) = 128$ in Fig. 2)
- $\phi(DSDG) = \lceil(72+6-3)\rceil = 75$
  (We calculate $\phi(DSDG) = 53$ in Fig. 2)

Using our new proposal, we have found it to decrease the complexities up to 59 percent as compared to the traditional *dynamic system dependence graph* technique. The slice results number using the traditional technique was 24, but the slice results number using our technique was only 19. It shows that our technique is efficient compared with the traditional technique because the complexities of the *dynamic system dependence graph* technique and the size of the sliced results have decreased.

5. Conclusion

The traditional slicing techniques execute a slicing through a *program dependence graph* or a *system dependence graph*. But, when a *system dependence graph* is drawn up, the complexities of the graph has increased because of addition of vertices as much as the number of parameters. The more the number of procedures and parameters, the more complex of the graph. In our approach, we have developed a *dynamic system dependence* technique to decrease the complexities of the graph. We have also illustrated our example with the C program environment to demonstrate that the

dynamic system dependence we proposed has the small slice size compared to a traditional *system dependence*. After we apply the dynamic system slicing algorithm to an example program in Figure 2 where a = {-2, 3}, we draw up the *dynamic system dependence graph* to compare to the *system dependence graph*. As the result, the complexities of the graph decrease 59 percent if we use the *dynamic system dependence graph* proposed in this paper. And the size of slice also decreases about 20 percent. In conclusion, we found that this DSDG approach has been more efficient compared with SDG.

References

[1] Mark Weiser, "Programmers use slices when debugging", Communications of the ACM, July 1982, pp.446-452.

[2] Mark Weiser. "Program slicing", IEEE Trans. on Software Engineering, July 1984, pp. 352-357.

[3] Reps, T. and Turnidge, T., "Program specialization via program slicing.", In Proceedings of the Dagstuhl Seminar on Partial Evaluation, 1996, pp. 409-429.

[4] Gupta R., Harrold M. and Soffa M., "An approach to Regression Testion Using Slicing.", Conf. software Maintenance, 1992, pp. 299-308.

[5] Korel B. and S. Yalamanchili, "Forward Derivation of Dynamic Slices.", Proc. Int'l Symp. Software Testing and Analysis, Seattle, 1994, pp.66-79.

[6] Kamkar, M., "Interprocedural Dynamic Slicing with Applications to Debugging and Testing.", PhD thesis, Linkoping Univ., 1993.

[7] Park, S. H. and Park, M. G., "An efficient dynamic program slicing algorithm and its Application.", Proc. of the IASTED International Conference, Pittsburgh, Pennsylvania, May 1998, pp.459-465.

[8] Jeanne Ferrante, Karl J. Ottentein, and Joe D. Warren. "The program dependence graph and its uses in optimization.", ACM Trans. on Programming Languages and Systems, July 1987, pp.319-349.

[9] Karl J. Ottentein and Linda M. Ottentein. "The program dependence graph in a software development environment.", Proc. of the ACM SIG SOFT/SIGPLAN Symposium on Practical Software Development Environments, Pittaburgh, Pennsylvania, April 1984.

[10] Susan Horwitz, Jan Prins, and Thomas Reps, "Integrating noninterfering versions of programs", ACM Trans. on Programming Languages and Systems, July 1989, pp.345-387.

[11] Susan Horwitz, "Identifying the semantic and textual differences between two versions of a program", Proc. of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, 1990, pp.234-245.

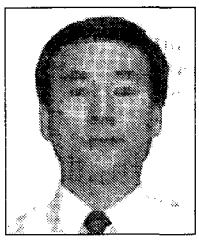
[12] Das, M. "Partial evaluation using dependence graphs.", Ph.D. dissertation and Tech. Rep. TR-1362, Computer Sciences Department, University of Wisconsin, Madison, WI, February 1998.

[13] Melski, D. and Reps, T., "Interprocedural path profiling", In Proc. of CC '99: 8th Int. Conf. on Compiler Construction, (Amsterdam, The Netherlands, Mar. 22-26, 1999), Lecture Notes in Computer Science, Vol. 1575, S. Jaehnichen (ed.), Springer-Verlag, New York, NY, 1999, pp.47-62.

[14] B. Korel, "Computation of Dynamic Program Slices for Unstructured Programs", IEEE Trans. On Software Engineering, vol. 23, No. 1, January 1997, pp.17-34.

박 순 형

1981년 울산대학교 공과대학 전자계산학과(학사)
1985년 숭실대학교 대학원 전자계산학과(석사)
1997년~현재 부경대학교 대학원 전자계산학과 박사과정 수료
1981년~1983년 현대미포조선(주) 전산실 근무
1987년~2000년 동의공업대학 전자계산과 교수
관심분야 : 소프트웨어공학 및 재공학, 소프트웨어 테스팅 , 비지니스 프로세스 재공학, 정보시스템 분석 및 설계, 비주얼 프로그래밍 기법, 멀티미디어 정보시스템 개발방법론



박 만 곤

1972년~1987년 경북대학교 학사/석사/박사 전산통계학(이학박사)
1993년 Univ. of Kansas 전기컴퓨터공학 (Post-Doc.)
1976년~1978년 육군 및 육군통신학교 통신장교 중위/대위
1978년~1980년 대구은행 사무전산부 전산개발주임
1979년~1981년 경남공업전문대학 전자계산학과교수
1987년~1990년 부경대학교 (구.부산수산대학교) 전자계산소장/자연과학대학 부학장
1988년~1997년 부경대학교 대학원/산업대학원/교육대학원 전자계산학과장 및 전공주임
1979년~2001년국제기구 Colombo Plan Staff College, 정보기술 및 정보통신학처장(외교통상부/과학기술부파견)
1997년~현재 ADB/UN ESCAP/ILO APSDEP/KOICA/JICA 국제정보기술자문교수
1981년~현재 부경대학교 공과대학 전자컴퓨터정보통신공학부 교수
1995년 University of South Australia 객원교수 (호주)
1992년~1993년 University of Kansas 교환교수(미국)
1990년~1991년 University of Liverpool 객원교수(영국)
관심분야 : 소프트웨어 엔지니어링, 소프트웨어 신뢰성 공학, 소프트웨어 품질공학, 소프트웨어 재사용 및 재공학, 멀티미디어정보처리기술, 멀티미디어 정보시스템, 멀티미디어 컨텐츠제작, 멀티미디어 코스웨어 개발, 멀티미디어 소프트웨어 공학, 인터넷 응용을 위한 멀티미디어 기술, 교육과 훈련을 위한 멀티미디어 활용
E-mail : mpark@pknu.ac.kr