JOURNAL OF KOREA MULTIMEDIA SOCIETY, VOL. 6, NO. 4, JULY 2003

Efficient Dynamic Object-Oriented Program Slicing

Soon-Hyung Park and Man-Gon Park

ABSTRACT

Traditional slicing techniques make slices through dependence graphs. They also 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, and size of the slices is larger. We propose the representation of a dynamic object-oriented program dependence graph so as to process the slicing of object-oriented programs that is composed of related programs in order to process certain jobs. We also propose an efficient slicing algorithm using the relations of relative tables in order to compute dynamic slices of object-oriented programs. Consequently, the efficiency of the proposed efficient dynamic object-oriented program dependence graph technique is also compared with the dependence graph techniques discussed previously. As a result, this is certifying that an efficient dynamic object-oriented program dependence graph is more efficient in comparison with the traditional object-oriented dependence graphs and dynamic object-oriented program dependence graph

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

1. INTRODUCTION

Program slicing is a progress of finding all statements in a program P that may directly or indirectly affect the value of a variable *var* at a point p. Accordingly, program slicing is a useful technique with other applications in program debugging by providing other programs that gather statements relating to an interested variable in a program[7,12,14]. Program slicing technique was proposed by Mark Weiser for the first time[10]. It has been suggested a usage of this concept in the program testing, maintenance, debugging, and program understanding. Object-oriented program slicing is working to get slices of object-oriented program by tracing the flow of classes that is the core of object-oriented program and objects. Generally it is important that in the object-oriented program slicing we present polymorphism, dynamic binding, class inheritance, etc[8].

Traditional program slicing techniques often use graphs as a process of slicing to generate correct slices[9,11]. But traditional dependence graphs especially object-oriented dependence graphs and dynamic object-oriented dependence graphs are complicated because that it need many vertexes and edges to represent data transmission inter procedures[6]. So it is very difficult that programmer and tester use them to debug source programs.

In this paper, we proposed several processes to compute the result of dynamic object-oriented dependence graph efficiently. We also demonstrated that this dynamic object-oriented program slicing technique is more effective than traditional object-oriented program slicing technique.

In section 2, we review the studies concerning traditional program slicing approaches. In section 3, we account for the Efficient Dynamic Object-orient Program Dependence Graph (EDOPDG) that

* *The authors are with the Dept. of Computer Science, Graduate School, PuKyong National Univ., 599-1 Daeyeon-Dong, Nam-Gu, Busan, Rep. of Korea.*
E-mails : nepaipark@hanmail.net and mpark@pknu.ac.kr
* *Dr. Man-Gon Park is also with the Colombo Plan Staff College (Inter-Governmental International Organization), Dep. of Ed. Complex, Meralco Ave., Pasig City, Metro Manila, Philippines.*
E-mail : mpark@cpsctech.org

is proposed in this paper. In section 4, we introduce the processes to compute dynamic object-oriented program slices. In section 5, we apply the processes for the application programs. The EDOPDG technique is compared with traditional methods in section 6.

2. DEFINITION OF SLICING

Program slicing is a course to generate program slices that is a set of statements that give effects to given variables directly or indirectly. The slicing technique is classified by the two criteria.

Firstly, it can be divided into static slicing and dynamic slicing by existence of execution history. Secondly, it can be divided into program slicing, system slicing and object-oriented program slicing by the number of programs that are objects of slicing[2,3,5].

Program slicing may be included the concept of system slicing. Especially, it may be called as procedure slicing where an object of the program slicing is single program. An important distinction of static slice and dynamic slice is that the former notion is computed without making assumptions regarding a programs input, whereas the latter relies on some specific test case[1,4].

```
CE1 :  class Elevator {
       public:
E2:    Elevator(int 1_top_floor)
S3:    {  current_floor = 1;
S4:       current_direction = UP;
S5:       top_floor = 1_top_floor; }
E6:    virtual ~Elevator() {}
E7:    void up()
S8:     { current_direction = UP; }
E9:    void down()
S10:    { current_direction = DOWN; }
E11:   int which_floor()
S12:    { return current_floor; }
E13:   Direction direction()
S14:    { return current_direction; }
E15:   virtual void go (int floor)
S16:    { if (current_direction == UP)
S17:      { while ((current_floor != floor) &&
                  (current_floor <= top_floor))
C18:        add(current_floor, 1); }
          else
S19:      { while ((current_floor != floor) &&
                   (current_floor > 0))
C20:        add(current_floor, -1); }
        }
       private:
E21:    add(int &a, const int& b)
S22:     { a = a + b; };
       protected:
        int current_floor;
        Direction current_direction;
        int top_floor;
       };
CE23:  class AlarmElevator : public Elevator {
       public:
E24:    AlarmElevator(int top_floor):
S25:      Elevator(top_floor)
S26:      { alarm_on = 0; }
E27:    void set_alarm()
S28:      { alarm_on = 1; }
E29:    void reset_alarm()
S30:      { alarm_on = 0; }
E31:    void go(int floor)
S32:     { if (!alarm_on)
C33:        Elevator::go(floor)
        } ;
       protected:
        int alarm_on;
       } ;
E34:    main(int argc, char **argv) {
        Elevator *e_ptr;
S35:    if (argv[1])
S36:      e_ptr = new AlarmElevator(10);
        else
S37:      e_ptr = new Elevator(10);
C38:    e_ptr->go(3);
```

```
S39:    cout << \n Currently on floor:
            << e_ptr->which_floor() << "\n";
        }
```

Fig. 1. Sample Program

3. EFFICIENT DYNAMIC OBJECT-ORIENTED PROGRAM DEPENDENCE GRAPH

An Efficient Dynamic Object-oriented Program Dependence Graph (EDOPDG) proposed in this paper is similar to the Program Dependence Graph (PDG) in the respect that the graphs represent the control dependence information by the control dependence edges and the data dependence information by the data dependence edges at the statements vertexes. The traditional object-oriented program dependence graphs is added the member variable edges, the call edges for construction of objects, the polymorphic call edges, the method call edges, etc. However, EDOPDG only is added the polymorphic call edges.

The process that is drawn up EDOPDG is as follows.

(1) We draw up edges in the graph using the static information of a source program within the limits of an execution history.

- class control dependence edges
- procedure control dependence edges
- method control dependence edges
- repetition control dependence edges
- selection control dependence edges
- inter-procedure edges
- return control dependence edges
- polymorphic choice edges
- polymorphic call edges
- polymorphic execution edges

(2) After we compute the data dependence edges, we add them to the graph if the paths of them in the graph are not already existent.

(3) After we compute the control dependence edges, we add them to the graph if the paths of

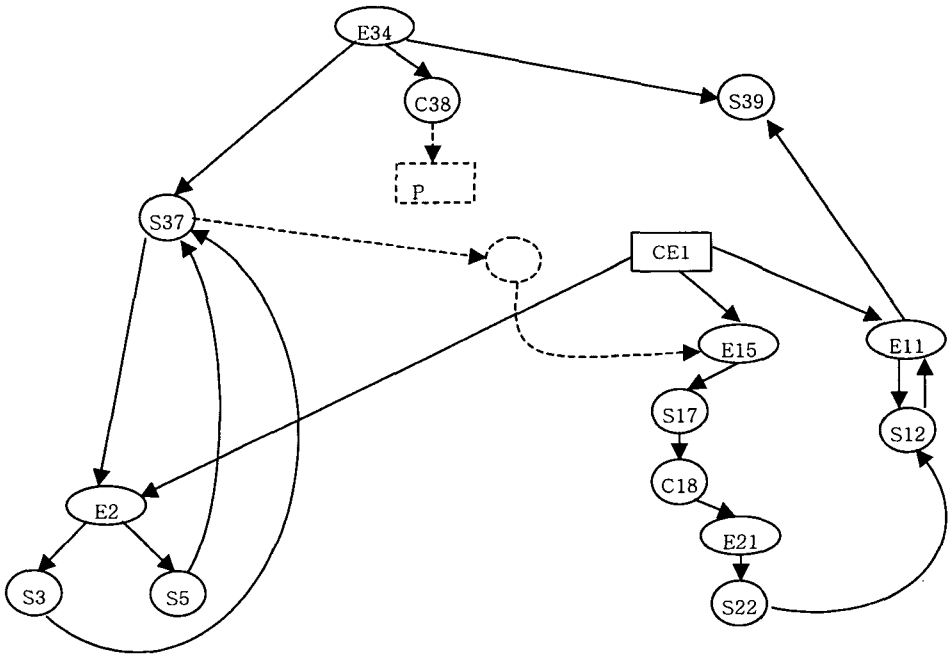

Fig. 2. The EDOPDG of sample program of Fig. 1.

them in the graph are not in existence. Start nodes of control dependence are as follows.

• selection control nodes that are in the upper level of nodes that are in existence two times and over in the area from the criterion node to the exit node of data dependence.

• repetition control nodes that are in the upper level of nodes that exist in the area from the criterion node to the exit node of data dependence.

4. STEPS OF THE DYNAMIC OBJECT-ORIENTED PROGRAM SLICING

The procedure that computes the dynamic object-oriented program slices using the efficient dynamic object-oriented program dependence graph (EDOPDG) is divided into four steps.

Firstly, a step of the program node analysis

Secondly, a step of the program execution history analysis

Thirdly, a step of the dynamic object-oriented program dependence graph generation

Finally, a step of the sliced program generation

An execution history is a set of the sequence $<v_1, v_2, ..., v_n>$ by order to be visited during execution of given test case.

4.1 A step of the program node analysis

A step of the program node analysis is a phase drawing up a table of related nodes on source programs. A table of the related nodes is a set of data that stores components of nodes of program statements. It is made up Node Numbers, Node types, DEFs, REFs, Upper position node and Upper repetition control node.

(1) Node types

Nodes that compose of programs are divided into 11 types.

(2) DEFs

A set of variables that have values changed at its node

(3) REFs

A set of variables that have values used at its nodes

(4) Upper position nodes

Upper position nodes of the current nodes

(5) Upper repetition control nodes

Upper repetition nodes of the current nodes

4.2 A step of the program execution history analysis

This step is a phase that analyzes source programs and draws up a execution history table when source programs are actually executed. An execution history table is a set of data on tracks operated when programs are executed. It is consisted of sequences of node execution and node numbers. Sequences of node execution imply the orders of execution history. Node numbers of the execution history table are equal to node numbers of a table of the related nodes.

4.3 A step of the dynamic object-oriented program dependence graph generation

This step is a phase that draws up EDOPDG by applying the efficient dynamic object-oriented program-slicing algorithm based on an execution history corresponding to the given input data.

The algorithm that computes dynamic object-oriented program slices (*Mark*) is as follows. We present the algorithm written in a let-in construct adapted from a similar construct in the programming language ML [13].

EDOPDG(<*prevhist* | *Mark*>) =
    SetVar' = *Ins*(*Criterion*, *SetVar*)
    DependCheck(*Criterion*, 1, 1)
    while k = *1*, *n*
        DependCheck(*IfCriterion*, *lastnum*, 2)
    end while

```
        while k = 1, m
            if SubSist(k, CheckObject)
            then DependCheck(RepeatCriterion, 1, 2)
            end if
        end while
    in (Criterion, IfCriterion, RepeatCriterion,
        CheckObject, Mark', SetVar')

DependCheck(Criterion, lastnum, RepeatUpper,
            CheckObject, init) =
    let startnum = Criterion
        while  k = startnum, lastnum, -1
            if (NodeType(num) = "R")
            then Return (Ref, num, SetVar, sist,
                         Mark, last),
            end if
            if (NodeType(num) = "A")
            then Assign(Mark, num, Def, Ref,
                        SetVar, last),
            end if
            if (NodeType(num) = "I")
            then Input(Mark, num, Def, SetVar, last),
            end if
            if (NodeType(num) = "CE" or
                NodeType(num) = "M" or
                NodeType(num) = "P" or
                NodeType(num) = "C" or
                NodeType(num) = "N")
            then Ins(Mark(i) , num),
                 last = i
            end if
            if (init = 1 and RepeatUpper(num) not
               = " ")
            then CheckObject(x) = num
            end if
            if (init = 1 and NodeType(num) = "D")
               Criterion(n) = (Ref, num),
                Select(Mark, num, Ref, SetVar, sist)
            end if
            if (init = 1 and NodeType(num) = "L")
               Criterion(m) = (Ref, num),
               Repeat(Mark, num, Ref, SetVar)
            end if
            end while
            lastnum = last
    in  (Criterion', lastnum', RepeatUpper
            CheckObject', init)

Return (Ref, num, SetVar, sist, Mark, last)
            SubSist(Ref(num), SetVar)
            if (sist = 1)
            then Ins(Mark(i) , num),
                 last = i
            end if
        in (Ref, num, SetVar', sist, Mark', last')

Assign(Mark, num, Def, Ref, SetVar, last)
        Ins(Mark(i), num),
        last = i,
        Del(Def(num), SetVar)
        if (Ref(num) not = " ")
        then Ins(Ref(num), SetVar)
        end if
    in (Mark', num, Def, Ref, SetVar', last')

Input(Mark, num, Def, SetVar, last)
         Ins(Mark(i), num),
         last = i,
         Del(Def(num) , SetVar)
    in (Mark', num, Def, SetVar', last')

Repeat(Mark, num, Ref, SetVar)
        SubSist(Ref(num), SetVar)
        if (sist = 1)
        then Ins(Mark(i), num),
             Ins(Ref(num), SetVar)
        end if
    in (Mark', num, Ref, SetVar')

Select(Mark, num, Ref, SetVar, sist)
        SubSist(Ref(num), SetVar)
        if (sist = 1)
        then Ins(Mark(i) , num),
             Ins(Ref(num), SetVar)
        end if
    in (Mark', num, Ref, SetVar', sist')
Ins(Var, SetVar) =
    Ins(Var, SetVar') =
```

$\cup_{x \in D} Var(x) \cup \cup_{x \in D} SetVar(x)$
in (*Var*, *SetVar'*)

Del(*Var*, *SetVar*) =
Del(*Var*, *SetVar'*) =
$\cup_{x \in D} SetVar(x) - \cup_{x \in D} Var(x)$
in (*Var*, *SetVar'*)

SubSist(*Var*(x), *SetVar*) =
let Sist = 0
if $(\cup_{x \in D} Var(x) \cap \cup_{x \in D} SetVar(x))$
then Sist = 1
end if
in (*Var*, *SetVar*, *Sist'*)

4.4 A step of the sliced program generation

This step is a phase that extracts program slices by traversing inversely EDOPDG to draw dynamic slices on based variable. A sliced program is a perfect program that can be executed for given input data.

5. APPLICATION EXAMPLE

We apply the dynamic object-oriented program slicing algorithm to an example program in Fig. 1 in order to make dynamic object-oriented slices where argv[1] = 3 and slicing criteria = (H, 39_{22}, which_floor). The types of nodes that consist in the program are noted in Table 1.

5.1 A step of the program nodes analysis

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

5.2 A step of the program execution history analysis

The execution history of the example program shown Fig. 1 is { 34, 35, 37, 2, 3, 4, 5, 38, 15, 16, 17, 18, 21, 22, 17, 18, 21, 22, 17, 11, 12, 39 } where

Table 1. Types of nodes

Type of node	Abbreviation	
class	CE	
method	E	M
procedure		P
call	S	C
return		R
assign		A
input		I
write		W
repeat		L
select		D
constructor		N

Table 2. The data of related nodes for the Example Program

NN	NT	DEF	REF	PMN
1	CE	Elevator		1
2	M	Elevator	1_top_floor	1
3	A	current_floor		2
4	A	current_direction		2
5	A	top_floor	1_top_floor	2
6	M	~Elevator		1
7	M	up		1
8	A	current_direction		7
9	M	down		1
10	A	current_direction		9
11	M	which_floor		1
12	R		current_floor	11
13	M	direction		1
14	R		current_direction	13
15	M	go	floor	1
16	D		current_direction	15
17	L		current_floor, floor, top_floor	15
18	C	add	current_floor	15
19	L		current_floor, floor	15
20	C	add	current_floor	15
21	P	add	a, b	18
22	A	a	a, b	21
23	CE	AlarmElevator		23
24	M	AlarmElevator	top_floor, current_direction	23
25	C	Elevator	top_floor	23
26	A	alarm_on		23
27	M	set_alarm		23
28	A	alarm_on		27
29	M	reset_alarm		23
30	A	alarm_on		29
31	M	go	floor	23
32	D		alarm_on	31

Table 2. Continued

NN	NT	DEF	REF	PMN
33	C	go	floor	31
34	E	main	argc, **argv	34
35	D		argv[1]	34
36	N	AlarmElevator		34
37	N	Elevator		34
38	C	go		34
39	W		which_floor	34

argv[1] = 3.

5.3 A step of the dynamic object-oriented program dependence graph generation

The EDOPDG of sample program shown Fig. 1 is illustrated in Fig. 2.

5.4 A step of the sliced program generation

A sliced program can be constructed by traversing the EDOPDG shown Fig. 2 to compute dynamic object-oriented program slicies where slicing criterion is which_floor of execution history order 39. The sliced program is illustrated in Fig. 3.

```
class Elevator {
 public:
   Elevator(int 1_top_floor)
   { current_floor = 1;
      top_floor = 1_top_floor; }
   int which_floor()
    { return current_floor; }
   virtual void go (int floor)
       { while ((current_floor != floor) &&
                (current_floor <= top_floor))
          add(current_floor, 1); }
   private:
   add(int &a, const int& b)
     { a = a + b; };
   protected:
    int current_floor;
    Direction current_direction;
    int top_floor;
   };
```

```
main(int argc, char **argv) {
 Elevator *e_ptr;
  e_ptr = new Elevator(10);
 e_ptr->go(3);
 cout <<"\n Currently on floor:"
       << e_ptr->which_floor() << "\n";
}
```

Fig. 3. Sliced program

6. EFFICIENCY ANALYSIS

The complexities of graph of the traditional object-oriented program dependence graph(OPDG), the traditional dynamic object-oriented program dependence graph(DOPDG) and the efficient dynamic object-oriented program dependence graph (EOPDG) proposed in this paper are all represented.

6.1 Complexities of the OPDG

The complexities of the traditional OPDG are represented below.

Type	Complexities of the OPDG
procedure dependence	pv + pc * (1 + pcp * 2) + p + 2 * pp + sc * (1 + scv * 2)
class dependence	s + (sv + sc) + m * 2 * (mp + scp)
Inter-class dependence	s * m * (1 + mp * 2)

Name of variable	Contents of variable
p	Procedure
pp	Parameter of procedure
pv	General vertex in the procedure
pc	Call in the procedure
pcp	Parameter of call in the procedure
sc	Class construction call
scv	Variable of class construction call
s	Class
m	Method
mp	Parameter of method
sc	Call in the class
scp	Parameter of call in the class
sv	Genaral vertax in the class

6.2 Complexities of the DOPDG

The complexities of the traditional DOPDG are represented below.

Type	Complexities of the DOPDG
procedure dependence	p + pc + 3 * pv
class dependence	s + sc + m + 3 * sv
Inter-class dependence	-

6.3 Complexities of the EDOPDG

The complexities of the traditional EDOPDG are represented below.

Type	Complexities of the EOPDG
procedure dependence	pv + p + 2 * (pc + sc)
class dependence	s + sv + sc + m
Inter-class dependence	-

6.4 Comparison of efficiency with OPDG

(1) The size of slices

The sizes of slices of the traditional OPDG techniques, the traditional DOPDG techniques and the EDOPDG technique proposed in this paper are represented below.

Type	Size of slices
OPDG	28
DOPDG	18
EDOPDG	15

The size of slices of the EDOPDG is smallest compared with that of the other graphs. Therefore we find that the technique of EDOPD is the best method among them to compute slices of the object-oriented programs.

(2) The comparison of complexities

The complexities of the traditional OPDG techniques, the traditional DOPDG techniques and the EDOPDG technique proposed in this paper are represented below.

Type	Maximum complexities	Actual complexities
OPDG	361	110
DOPDG	69	22
EDOPDG	42	17

The value of maximum complexities in the traditional OPDG is different from that of actual complexities because all of the parameters of call statements may not be able to be changed. In the case of the repetition statements, the value of maximum complexities in the traditional DOPDG is different from that of actual complexities on account of the number of repetition nodes. The value of maximum complexities in the EDOPDG is different from that of actual complexities because of vertexes that are not contained in the dynamic slices.

The value of the complexities of the EDOPDG is smallest comparing with that of the other graphs.

7. CONCLUSION

Static slices are a set of nodes that affect criterion variables. Dynamic slices are a set of nodes that affect actually the values of variables tracing on the test case. Therefore we can use usefully a dynamic concept in the field of the debugging through a test case.

We propose a dynamic object-oriented slicing technique using EDOPDG in this paper. We find that the complexities of the EDOPDG is 42, the traditional complexities of the OPDG is 361 and the traditional complexities of the DOPDG is 69 with a result that we apply an example program of the

fig. 1 to the formulas of the complexities using the traditional OPDG technique, the traditional DOPDG technique and the EDOPDG technique. As the result, the values of the actual complexities of the EDOPDG, OPDG and DOPDG are 17, 110 and 22 respectively.

The size of the slices of the EDOPDG is 15 where the slicing criterion is which_floor in the node 39. The sizes of the slices of the OPDG and DOPDG are 28 and 18 respectively.

We find that the approach of the EDOPDG is more efficient compared with those of the OPDG and DOPDG.

8. REFERENCES

[1] Arpad Beszedes, Tamas Gergely, Zsolt Mihaly Szabo, Janos Csirik, Tibor Gyimothy, "Dynamic Slicing Method for Maintenance of Large C Programs.", Conference on Software Maintenance and Reengineering (CSMR), Lisbon, Portugal, pp.105-113, 2001.

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

[3] B. Korel, "Computation of dynamic slices for unstructured programs.", IEEE Transactions on Software Engineering, vol.23, No.1, pp.17-34, 1997.

[4] B. Korel and J. Laski, "Dynamic Program slicing.", Information Proceeding Letters, vol.29, No.3, pp.155-163, 1998.

[5] Hiralal. Agrawal and J. R. Horgan, "Dynamic Program Slicing.", Proc. ACM SIGPLAN'90 Conf. Programming Lang. Design and Implementaion, pp.246-256, 1990.

[6] J. Zhao, "Dynamic Slicing of Object-Oriented Programs," Technical-Report SE-98-119, pp. 17-23, Information Processing Society of Japan (IPSJ), May 1998.

[7] 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, Pennsilvania, April 1984.

[8] Loren D. Larsen and Mary Jean Harrold, "Slicing Object-Oriented Software.", Technicdal Report 95-103, Department of Computer Science, Clemson University, March 1995.

[9] Margaret Ann Francel, Spencer Rugaber, "The value of slicing while debugging.", Science of Computer Programming, Volume 40, Number 2-3, pp.151-169, July 2001.

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

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

[12] Raghavan Komondoor, Susan Horwitz, "Tool Demonstration: Finding Duplicated Code Using Program Dependences.", European Symposium on Programming (ESOP), Genova, Italy, pp.383-386, 2001.

[13] Robin Milner, Mads Tofte, and Robert Harper. "The Definition of Standard ML.", The MIT Press, Cambridge, MA, 1990.

[14] Susan Horwitz, T. Reps and David Binkley. "Interprocedural Slicing using Dependence Graph.", ACM Tran. on Programming Languages and Systems, vol. 12, no. January 1990.

Soon-Hyung Park

Dr. Soon-Hyung Park is a Lecturer of the Department of Computer Science at PuKyong National University (PKNU), Korea.

Also Dr. Soon-Hyung Park is serving for the Inter-Governmental International Organization, Colombo Plan Staff College as an assistant faculty.

From 1987 to 2002, he was a professor of the DongEui Institute of Technology, Busan, Korea and also he worked for Hyundai-Mipo Dockyard Co., Ulsan City, Rep. of Korea as a Computer programmer from 1981 to 1986.

He received his bachelors, masters and doctoral degrees in Computer Science from the Ulsan University, Soongsil University and PuKyong National University, Rep. of Korea, respectively.

His interesting fields in research are: software testing, program slicing and merging, software reengineering, multimedia information processing technology, and Quality Management System.

Man-Gon Park

Prof. Man-Gon Park is a Professor of the Department of Computer Science at PuKyong National University (PKNU), Korea, where he had worked since 1981. He was also visiting professor at the Department of Computer Science, University of Liverpool, UK; exchange professor at the Department of Electrical and Computer Engineering, University of Kansas, USA; and visiting scholar at the School of Computers and information science, University of South Australia. He was dispatched to Mongolia and China by KOICA on various projects as information systems consultant. He has joined in international consulting works for Sri Lanka, Vietnam, and other countries funded by ADB, ILO and other international organizations.

Currently Dr Man-Gon Park is holding the Director and CEO of the Inter-Governmental International Organization, Colombo Plan Staff College, concurrently with a PKNU professor as the principal position by Korean Government.

Some of his areas of interest in research are: software reliability engineering, business process reengineering, Internet & Web Technology, and multimedia information processing technology, and Quality Management System. He is a member and academic board member of such professional societies as the KIPS, the Korean Multimedia Society, IEEE, ACM and IASTED. He has authored and co-authored lots of academic and technical papers, books and other reports in his field of expertise. He received his bachelors, masters and doctoral degrees in Statistical Computing Science from the KyungPook National University, Korea.

Recently Korean Government has awarded him with the official commendation for his professional contributions and achievements in development of knowledge networking system & web-based information systems.

For information of this article, please send e-mail to: mpark@pknu.ac.kr(Man-Gon Park)