# Slicing Methods Using Static and Dynamic Analysis Information

Yoshiyuki Ashida[†], Fumiaki Ohata[†] and Katsuro Inoue[†,††]
[†] Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan
{asida, oohata, inoue}@ics.es.osaka-u.ac.jp

[† †] Graduate School of Information Science, Nana Institute of Science and Technology

## Abstract

*In this paper, we propose four slicing methods using both static and dynamic analysis information. (1) Statement-Mark Slice : removes the unnecessary statements using an execution history of the statements. (2) Partial Program Analysis : reduces the static analysis cost using invocation history of procedures. (3) Dynamic Data Dependence Analysis : extracts precise data dependence relations using dynamic data dependence analysis. (4) Array and Pointer Analysis : improves the efficiency of (3) by dynamically analyzing pointer and array variables only. Using both dynamic and static information, we will show that the precision of the slicing is improved with smaller run-time overhead.*

## 1. Introduction

Program slice[14] is a set of statements that affect the value of variable $v$ in a statement $s$. In order to calculate a program slice, we must know the dependence relations between statements in the program.

Program slicing is very promising approach for program debugging, testing, understanding, merging, and so on[2, 3, 5, 7, 14]. We have empirically investigated effectiveness of program slicing for program debugging and program maintenance processes, and its significance was validated by several experiments [9, 10].

Program slicing techniques are roughly divided into two categories, static slicing [14] and dynamic slicing [1]. The former is based on static analysis of source program without input data. The dependence of program statements is investigated for all possible input data. The latter is based on a specific input data, and the dependence of the program statements is explored for the program execution with the input data. The size of the static slice is larger in general, since it considers all possible input data. The size of the dynamic slice is smaller in general, but it requires a large amount of CPU time and memory space to obtain it.

When we focus on the program debugging, we believe that the dynamic slice is more effective for program debugging than the static one, since the debugging process often needs program executions. However, computing dynamic slicing associated with program execution is very expensive. We thought that using both static and dynamic information might be better than using only dynamic information.

In this paper, we propose four techniques using the static and dynamic information.

**(1)** Statement-Mark Slicing makes a PDG from a source program and its execution history.

**(2)** Partial Analysis uses a source program and invocation history of call-statement.

**(3)** Dynamic Data Dependence Analysis gets data dependences while execution and control dependences from source program.

**(4)** Array and Pointer Analysis is similar to (3). This method obtains data dependences of only array and pointer variables while execution.

In section 2, we will briefly overview program slice. In section 3, we will present Statement-Mark slice and evaluate it. In section 4, we will present Partial Analysis and evaluate it. In section 5, we will present Dynamic Data Dependence Analysis. In section 7, we will conclude our discussions with a few remarks.

## 2. Program Slice

### 2.1. Static Slice

Consider statements $s_1$ and $s_2$ in the source program $p$. When all of the following conditions are satisfied, *a control dependence* , CD, from $s_1$ to $s_2$ exists :

- $s_1$ is a conditional predicate, and
- the result of $s_1$ determines whether $s_2$ is executed or not.

This relation is written by '$CD(s_1, s_2)$'.

When the following conditions are all satisfied, *a data dependence* , DD, from $s_1$ to $s_2$ exists.

- $s_1$ defines $v$, and
- $s_2$ refers $v$, and
- at least one execution path from $s_1$ to $s_2$ without redefining $v$ exists.

This relation is denoted by '$DD(s_1, v, s_2)$'.

In order to slice the program, we commonly use '*Program Dependence Graph* (PDG)'. A PDG is a directed graph whose nodes represent statements in a source program, and whose edges denote dependence relations(DD or CD) between statements(nodes). A DD edge is labeled with a variable name '$a$' if it denote DD($\cdots$, $a$, $\cdots$). An edge drawn from node $V_s$ to node $V_t$ represents that 'node $V_t$ depends on node $V_s$'. Fig.2 shows PDG of Fig.1.

Then we calculate a static slice with *a slicing criterion*(a pair $(s, v)$, $s$ is a statement and $v$ is a variable used, defined or referred in $s$).

In order to get a slice for slicing criterion $(s, v)$, PDG nodes are traversed inversely from $V_s$(node $V_s$ denotes statement $s$.). The reached nodes from $V_s$ with respect to variable $v$ and other transitive variables form a slice for $(s, v)$. Fig.3 shows a slice(underlined statements) of slicing criterion $(24, d)$ for the program shown in Fig.1.

```
1   program Square_Cube(input,output);
2   var a,b,c,d : integer;
3   function Square(x : integer):integer;
4   begin
5       Square := x*x
6   end;
7   function Cube(x : integer):integer;
8   begin
9       Cube := x*x*x
10  end;
11  begin
12      writeln("Squared Value ?");
13      readln(a);
14      writeln("Cubed Value ?");
15      readln(b);
16      writeln("Select Feature! Square:0, Cube: 1");
17      readln(c);
18      if(c = 0) then
19          d := Square(a)
20      else
21          d := Cube(b);
22      if (d < 0) then
23          d := -1 * d;
24      writeln(d)
25  end.
```

**Figure 1. Pascal Source Program**

### 2.2. Dynamic Slice

In a calculation of a static slice, we make a dependence analysis in a source program. In a calculation of a dynamic slice, we analyze a dependence from an execution history. This history records the execution of statements as the program executes. And one execution in an execution history is called execution point.

Consider two execution points $r_1$ and $r_2$. When all of the following conditions are satisfied, *a dynamic control dependence,* DCD, from $r_1$ to $r_2$ exists :

- $r_1$ is a conditional predicate, and
- the result of $r_1$ determines whether $r_2$ is executed or not.

This relation is written by '$DCD(r_1, r_2)$'.

When the following conditions are all satisfied, *a dynamic data dependence* , DDD, from $r_1$ to $r_2$ exists.

- $r_1$ defines $v$, and
- $r_2$ refers $v$, and
- no execution between $r_1$ and $r_2$ defines $v$.

This relation is denoted by '$DDD(r_1, v, r_2)$'.

In dynamic slicing, we use '*Dynamic Dependence Graph*(DDG)'. A DDG is a directed graph whose nodes represent execution points, and whose edges denote dependence relations(DDD or DCD) between execution points(nodes).

Then we specify an input $x$, an execution point $r$ and a variable $v$ as a slicing criterion, and DDG nodes are traversed inversely for slicing criterion $(x, r, v)$. Finally, the result on DDG is mapped onto the source list.

Fig.4 shows a slice of slicing criterion $(x, 13, d)$ for the program shown in Fig.1(input x is $a = 2$, $b = 3$, $c = 0$, and at execution point 13, statement 24 is executed).

### 2.3. Features of Static and Dynamic Slice

When we calculate a static slice, we use PDG based on the dependence analysis of source program. The cost of

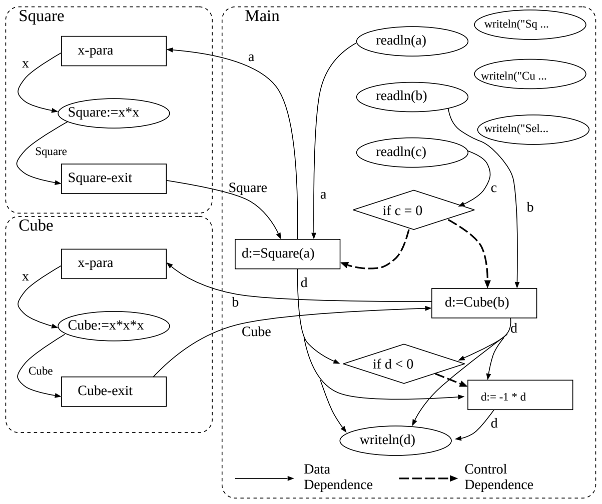


**Figure 2. PDG**

```
1    program Square_Cube(input,output);
2    var a,b,c,d : integer;
3    function Square(x : integer):integer;
4    begin
5        Square := x*x
6    end;
7    function Cube(x : integer):integer;
8    begin
9        Cube := x*x*x
10   end;
11   begin
12       writeln("Squared Value ?");
13       readln(a);
14       writeln("Cubed Value ?");
15       readln(b);
16       writeln("Select Feature! Square:0, Cube: 1");
17       readln(c);
18       if(c = 0) then
19           d := Square(a)
20       else
21           d := Cube(b);
22       if (d ¡ 0) then
23           d := −1 ∗ d;
24       writeln(d)
25   end.
```

**Figure 3. Static Slicing Result by $d$ at Line 24**

constructing PDG is relatively small[12, 13], but the size of the slice is relatively large because of considering all possible inputs.

On the other hand, the dynamic slice is calculated from DDG based on the dependence analysis of the execution history. Thus, statements that have not executed is removed from slice, and the size of dynamic slice is smaller than that of static.

If a program fails in a specific input data, dynamic slice is very useful to find the fault which causes the failure.

In order to calculate dynamic slice, while static analysis before execution is not needed, we must know dynamic dependence relations while execution, requiring large memory space and execution overhead. This worsens the efficiency of debugging process.

## 3. Statement-Mark Slice

In this section, we show *Statement-Mark slicing* which uses the information of which statement has executed.

### 3.1. Call-Mark Slice

We have already proposed the *Call-Mark Slicing*[8, 11] which is between the static slicing and the dynamic slicing. A call-mark slice is obtained as follows:

1. In the same way of the static slicing, a PDG is constructed from a source program.

2. The program is executed with an input data, and each call statements was marked whether they were executed or not.

3. Using marked call statements, we specify unexecuted statements, and remove them from the static slice.

Since we need to mark only call statements, a call-mark slice needs less execution time and memory space than the dynamic slicing.

### 3.2. Statement-Mark Slice

In call-mark slicing, we need to mark call statements. If we mark all statements, we would expect to get more precise slice than the call-mark slice.

The related idea has been introduced in [4], and we will discuss the implementation and evaluation of this method in this section.

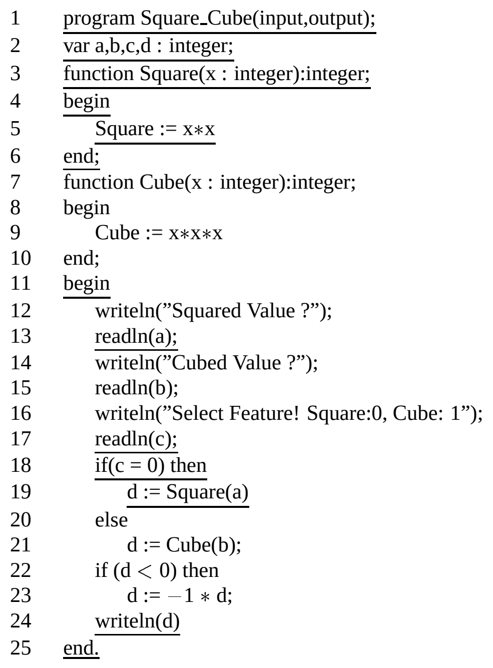
**Method of Statement-Mark Slicing**

The basic way of statement-mark slicing is same as the call-mark slicing:

1. A PDG is constructed from a source program.

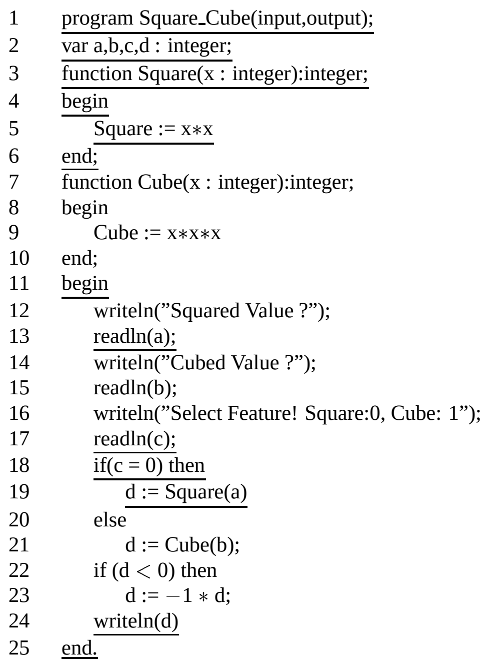
2. The program is executed with an input data, and all the statements were marked whether they were executed or not.

3. PDG nodes are traversed inversely from slicing criterion. If it reaches the unexecuted node, we remove



```
1   program Square_Cube(input,output);
2   var a,b,c,d : integer;
3   function Square(x : integer):integer;
4   begin
5       Square := x∗x
6   end;
7   function Cube(x : integer):integer;
8   begin
9       Cube := x∗x∗x
10  end;
11  begin
12      writeln(”Squared Value ?”);
13      readln(a);
14      writeln(”Cubed Value ?”);
15      readln(b);
16      writeln(”Select Feature! Square:0, Cube: 1”);
17      readln(c);
18      if(c = 0) then
19          d := Square(a)
20      else
21          d := Cube(b);
22      if (d < 0) then
23          d := −1 ∗ d;
24      writeln(d)
25  end.
```

**Figure 4. Dynamic Slicing Result by $d$ at Line 24 with input ($a = 2$, $b = 3$, $c = 0$)**



```
1   program Square_Cube(input,output);
2   var a,b,c,d : integer;
3   function Square(x : integer):integer;
4   begin
5       Square := x∗x
6   end;
7   function Cube(x : integer):integer;
8   begin
9       Cube := x∗x∗x
10  end;
11  begin
12      writeln(”Squared Value ?”);
13      readln(a);
14      writeln(”Cubed Value ?”);
15      readln(b);
16      writeln(”Select Feature! Square:0, Cube: 1”);
17      readln(c);
18      if(c = 0) then
19          d := Square(a)
20      else
21          d := Cube(b);
22      if (d < 0) then
23          d := −1 ∗ d;
24      writeln(d)
25  end.
```

**Figure 5. Statement-Mark Slicing Result by $d$ at Line 24 with input ($a = 2$, $b = 3$, $c = 0$)**

this node from slice, stop traversing, and find another branch.

Fig.5 shows a slice of slicing criterion $(24, d)$ for the program shown in Fig.1(input is $a = 2$, $b = 3$, $c = 0$).

In this case, the statement-mark slice is same as the dynamic slice. A statement-mark slice becomes the superset of the dynamic slice.

**Evaluation of Statement-Mark Slicing**

In order to validate the statement-mark slicing, we have implemented this method within our Osaka Slicing System(OSS)[12]. And then, we have measured the size of slice and the execution time. Tab.1 and Tab.2 show the results.

In comparison with the call-mark slice, the size of the statement-mark slice is 10–30% smaller, and the execution time is 15–30% longer.

In comparison with static slice, its size is 20–55% smaller, and the execution time is 30–60% longer.

We would think that this approach is a very good compromise of slice precision and slice cost. The static slicing is low precision and low cost, and the dynamic slicing is high precision and high cost. The call-mark slicing and statement-mark slicing are between them. The statement-mark slicing produces more precise results but requires more run-time overhead.

**Table 1. Size of Various Slicing Results(LOC)**

| program | static | call-mark | statement -Mark | dynamic |
|---|---|---|---|---|
| P1 | 27 | 19 | 15 | 14 |
| P2 | 176 | 155 | 141 | 139 |
| P3 | 324 | 166 | 148 | 50 |

(Pentium-II 300MHz with 256MB Memory)

## 4. Partial Analysis of Source Program

The static slicing does not consider of the input data. However, if we execute a source program with an input data, we would get information for improvement of static analysis.

We need not to analyze unexecuted statements, and will reduce both the size of slice and the cost of constructing PDG.

Here, we propose the *Partial Analysis method* as the improvement on the call-mark slicing.

### 4.1. Partial Analysis Method

A static slicing algorithm proposal in [13] is divided two phases:

**Table 2. Execution Time(ms)**

| program | static | call-mark | statement -Mark | dynamic |
|---|---|---|---|---|
| P1 | 38 | 47 | 62 | 87 |
| P2 | 48 | 53 | 62 | 903 |
| P3 | 4,064 | 4,104 | 5,318 | 31,635 |

(Pentium-II 300MHz with 256MB Memory)

**Table 3. Analysis Time(ms)**

| program | static | call-mark | partial analysis |
|---|---|---|---|
| P1 | 21 | 22 | 13 |
| P2 | 1,602 | 1,625 | 1,142 |
| P3 | 8,125 | 8,207 | 3,957 |

(Pentium-II 300MHz with 256MB Memory)

**(A)** Intraprocedural analysis

**(B)** Interprocedural analysis

The partial analysis is a method improved on the above algorithm as follows:

1. Execute the program with an input data, and mark call statements whether they were executed or not.

2. Make an intraprocedural analysis. On that occasion, uncalled procedures are not analyzed.

3. Make an interprocedural analysis. Unexecuted call statements are not analyzed.

This partial analysis uses the call-mark information. If we use statement-mark information, we can expect more precise slice with extra cost. To perform the partial analysis using statement-mark information, the execution time and the size of slice will be almost the same as the statement-mark slice.

## 4.2. Evaluation of Partial Analysis

Like the Statement-Mark slicing, we have implemented this method within our OSS. And then, we have measured the analysis time. Tab.3 shows the result.

Not analyzing the unexecuted call statements and the uncalled procedures, the analysis time reduced 30–50% from the static and the call-mark analysis.

## 5. Dynamic Data Dependence Analysis

### 5.1. Array and Pointer Analysis

When we make data dependence analysis in an array variable, it is very difficult to know the value of array indices, and we get a lot of unwilling data dependence relation.

```
1  a[0]:=0;
2  a[1]:=1;
3  a[2]:=2;
4  readln(c);
5  b:=a[c]+5;
6  writeln(b);
```

**Figure 6. Pascal Program Including An Array Variable**

```
1  a=2;
2  b=1;
3  c=&a;
4  d=&c;
5  *c=5;
6  **d=b;
7  printf("%d",a);
8  printf("%d",**d);
```

**Figure 7. C Program Using Pointers**

Fig.6 is a simple program including an array variable. In the static analysis, we can not get the input value(to $c$) at statement $4$, and therefore we conclude that statement $5$ depends on all of statements 1–3.

In the case of presence of pointer, implicit data dependences emerge because of aliases by the pointers. We need high cost to compute safe approximation of the data dependences, it is impractical to analyze the dependence relations statically.

Fig.7 is a simple program using pointers. Variable $a$ at statement $7$ is defined at statement $6$, although analyzing this dependence relation is very difficult.

### 5.2. Overview of Analysis

Static analyses of array and pointer variables need much cost and produces results of low precision. Then we propose *Dynamic Data Dependence Analysis*. This method has following features.

1. Data dependence analysis is made dynamically.

2. Control dependences are computed statically(not DCD).

3. Nodes in the graph represent statements in a source program (The dependence graph is a PDG, not DDG).

Because of 1, the slice size will be close to that of the dynamic slice, and due to 2 and 3, the execution time is shorter than that of dynamic.

When a variable $v$ is referred at statement $s$ during an execution, if we know statement $t$ defines $v$ just before, we say that $DD(t, v, s)$ exists.

Then, if we save the information for all the variables which statement defined their values, we can obtain the precise data dependence relation even if there are array and pointer variables.

## 5.3. Analysis

A statement $s$ has the set named $DDS(s)$. An element of $DDS(s)$ is a tuple: ("a variable referred at $s$", "a statement on which $s$ depends").

At a certain execution point $p$, we denote $DefS(v)$ as the statement which defined $v$ just before $p$.

**(1)** Before execution, assign $\phi$ to $DDS(s)$ for all statement $s$.

**(2)** Execute the program with an input data. Assume that $s$ is now executed.

- if $v$ has referred at $s$, $(v, DefS(v))$ is added to $DDS(s)$.
- if $v$ has defined at $s$, $s$ is assigned to $Defs(v)$.

After all, $DDS(s)$ is equal to the data dependence relation to $s$, it means $DDS(s) = \{(v, t) | DD(t, v, s)$ holds.$\}$.

**(3)** Constructing the PDG from $DDS$. In the analysis of $s$,

- While $DDS(s) \neq \phi$, repeat the following two operations.
  1. $DDS(s) \leftarrow DDS(s) - \{(v, t)\}$.
  2. Draw a DD edge from $t$ to $s$ with $v$.
- If $s$ is a conditional predicate, draw CD edges from $s$ to predicate $t$.

Tab.4 shows the transition at each execution point for the program shown Fig6.

The statement 1 defines variable $a[0]$, and it causes $DefS(a[0]) = 1$. At the statement 2, 3, 4 and 5, $DefS$ is assigned as well as statement 1.

Since statement 5 refers $a[0]$, $(a[0], DefS(a[0]))$ is added to $DDS(5)$. Finally it becomes $DDS(5) = \{(a[0], 1), (c, 4)\}$.

## 6. Dynamic Analysis for Arrays and Pointers

Since the former method needs to be consider $DefS$ for all variables, there is a high possibility to need a large memory space and a long execution time.

**Table 4. Transition of $DefS$ at Each Execution Point in Program Fig.6**

| statement | $a[0]$ | $a[1]$ | $a[2]$ | $b$ | $c$ |
|---|---|---|---|---|---|
| 1 | 1 | – | – | – | – |
| 2 | 1 | 2 | – | – | – |
| 3 | 1 | 2 | 3 | – | – |
| 4 | 1 | 2 | 3 | – | 4 |
| 5 | 1 | 2 | 3 | 5 | 4 |
| 6 | 1 | 2 | 3 | 5 | 4 |

Originally, that method aims at reducing slice size with precise array and pointer analysis. Therefore, there is another option such that the dynamic analysis is applied to arrays and pointers only, and the static analysis is used for other variables.

In this approach, we must consider all array variables and also variables that can be pointed by the pointer. Here, we consider languages whose pointers can point to limited objects such as dynamically created variables.

We describe only the difference between this approach and the former method.

- Consider $DefS$, for each elements of arrays and the variables that can be pointed.
- During execution, if array and pointer reference occurs, define $DDS$ using $DefS$.
- After execution, static analysis is made. Arrays and pointers are analyzed using $DDS$, and an normal static analysis is done for other variables.

## 7. Conclusion and Future Work

We have presented four slicing methods using static and dynamic information.

In statement-mark slice, we have implemented this method and have evaluated it. As a result, we have noticed that the statement-mark slice is 20–55% smaller, and the execution time is 30–60% longer than those of the static slice.

In partial analysis, we have also implemented and evaluated it, and we know that the analysis time was reduced by 30–50% from the static analysis.

In dynamic data dependence analysis, we have presented the overviews of the approach. The implementation is a future theme.

By combining the static information and the dynamic information, we have shown that we can obtain suitable compromises of slice precision and slicing performance. The methods presented here are very useful and promising approach to construct practical slicing tools.

Also, we have extended the idea of using dynamic information to the non-scalar type variables such as array and pointer variables.

Guputa et al. proposed Hybrid Slicing[6], where they use both static and dynamic information. In the hybrid slicing, trace history of break points and procedure call/return is used. On the other hand, we use only one bit flag for execution of each statement, which is more simply implemented in the slicing system.

The statement-mark slicing corresponds to a simplified approach of the dynamic slicing proposed by Agrawal and Horgan[1]. We have here presented a practical implementation method and also shown the effectiveness of this approach with comparison to the static and dynamic slicing.

We are planning as follows:

- Implementation of the dynamic data dependence analysis
- Evaluation of our methods for large programs

## 8. Acknowledgements

This work is partly supported by Ministry of Education, Science, Sports, and Culture, grant-in-aid for priority areas "Principles for Constructing Evolutionary Software", #09245106.

## References

[1] Agrawal, H. and Horgan, J. : "Dynamic Program Slicing", *SIGPLAN Notices, Vol.25, No.6*, pp. 246–256, 1990.

[2] Bates, S. and Horwitz, S. : "Incremental Program Testing Using Program Dependence Graphs", *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, 1993.

[3] Beck, J. and Eichmann, D. : "Program and Interface Slicing for Reverse Engineering", *Proceedings of the 15th International Conference on Software Engineering*, pp. 509–518, 1993.

[4] Binkley, D.W. and Gallagher, K.B. : "Program Slicing", *Advances in Computers, Volume 43*, Marvin Zelkowitz, Editor, Academic Press San Diego, CA, 1996.

[5] Gallagher, K.B. and Lyle, J.R. : "Using Program Slicing in Software Maintenance", *IEEE Transactions on Software Engineering, 17(8)*, pp. 751–761, 1991.

[6] Gupta, R., Soffa, M.L., and Howard, J. : "Hybrid Slicing: Integrating Dynamic Information with Static Analysis", *ACM Transaction on Software Engineering and Methodology, Vol. 6, No. 4*, pp. 370–397, 1997.

[7] Horwitz, S. and Reps, T. : "The Use of Program Dependence Graphs in Software Engineering", *Proceedings of the 14th International Conference on Software Engineering*, pp. 392–411, 1992.

[8] Jihira, M., Nishimatsu, A., Kusumoto, S. and Inoue, K. : "Program Slicing Technique Using Function Call History", *Technical Report of IEICE, SS98-7*, pp. 9–15, 1998.(in Japanese)

[9] Nishie, K., Kamiya, T., Kusumoto, S. and Inoue, K. : "Experimental Evaluation of Usefulness of Debugging Support System Based on Program Slicing", *Proceedings of Software Symposium '97*, pp. 142–147, Japan, 1997(in Japanese).

[10] Nishimatsu, A., Kusumoto, S. and Inoue, K. : "An Experimental Evaluation of Program Slicing on Fault Localization Process", *Technical Report of IEICE, SS98-3*, pp. 17–24, Japan, 1998(in Japanese).

[11] Nishimatsu, A., Jihira, M., Kusumoto, S. and Inoue, K. : "Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice", *Proceedings of The 21st International Conference on Software Engineering*, Los Angeles, CA, USA, 1999.

[12] Sato, S., Iida, H., and Inoue, K. : "Software Debug Supporting Tool Based on Program Dependence Analysis", *Transaction on IPSJ, Vol. 37, No. 4*, pp. 536–545, Japan, 1996(in Japanese).

[13] Ueda, R., Ren, R., Inoue, K. and Torii, K. : "An Algorithm of Computing Slices for Recursive Program", *Transaction on IEICE, Vol. J78-D-I, No. 1*, pp. 11–22, 1995(in Japanese).

[14] Weiser, M. : "Program Slicing", *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439–449, 1981.