Systems and Computers in Japan, Vol. 32, No. 10, 2001
Translated from Denshi Joho Tsushin Gakkai Ronbunshi, Vol. J82-D-I, No. 10, October 1999, pp. 1256–1264

A Slice Reduction Method Using Function Activation Information

Akira Nishimatsu,[1] Minoru Jihira,[2] Shinji Kusumoto,[1] and Katsuro Inoue[1]

[1]Graduate School of Engineering Science, Osaka University, Toyonaka, 560-8531 Japan

[2]Department of Information Science, Nara Institute of Science and Technology, Ikoma, 630-0101 Japan

SUMMARY

During debugging or maintenance of large-scale software, it is useful to be able to restrict the reference range to a part of the program rather than reference the entire program. Program slicing has been proposed as a method to restrict the reference range. A static slice extracts the set of statements that can affect the values of the variables being watched. In many cases, however, the slices include parts that are in fact unrelated, and so the reference range cannot be sufficiently restricted. On the other hand, a dynamic slice can sufficiently restrict the reference range by actually executing the program and then using the dynamic information thus obtained to extract the set of statements which, when executed, may have an effect on the value of the variables being watched. However, considerable time and space are required in order to obtain the dynamic information during program execution. In this paper the authors propose a program slice extraction method which combines the small amount of dynamic information obtained by executing the program with static analysis. Slices extracted using this method are referred to as Call-Mark slices. When calculating a Call-Mark slice, first the data dependence relationship and the control dependence relationship in the program are analyzed in a static manner, then the procedures and function call memory during program execution are stored. Then, by clarifying the dynamic dependence relationship in the variables from the above information, slices can be collected more efficiently than in conventional approaches. In addition, the authors use the Call-Mark slice in a slicing system that was previously developed, and then evaluate its validity. © 2001 Scripta Technica, Syst Comp Jpn, 32(10): 59–68, 2001

Key words: Static slice; dynamic slice; dependence analysis; execution overhead.

1. Introduction

Debugging and maintenance of large-scale software represents an important topic in research in software engineering. One approach for debugging and maintaining large-scale software efficiently is to limit the reference range by extracting the parts which are directly or indirectly related to the information to be known, and not reference the entire program. A method using a program slice (hereafter, "slice") has been proposed for this approach [8, 19].

The authors have already experimentally evaluated the validity of slices for debugging and maintenance [14]. In that experiment, subjects were divided into two groups. The subjects in the first group used only conventional debugging tools, whereas those in the second group found the locations of faults using conventional debugging tools and slice tools. The time required for each group to find the locations of the faults was then measured. A statistical analysis of the required time revealed a meaningful difference between the working time for the two groups. The use

of slices restricted the range which the program referenced. Concentrating only on the restricted range improved efficiency. As a result, finding the location of faults could be performed more efficiently.

Considerable research has already been done on slices [7, 8]. A static slice is a slice for which a set of statements that may affect the values of variables being focused on is extracted. In general, a static slice is a technology for extracting a part of a program from the original program. During source code analysis, the size of the extracted slice is often large because of consideration of all input and flow control. A dynamic slice is a slice for which a set of executed statements that may affect the values of variables being focused on is extracted with respect to program execution in which data are provided [1, 9].

Because a dynamic slice is based on execution with respect to a particular input, a statement dependent only on a statement which is not executed, and a statement which is not executed are automatically excluded, and thus a dynamic slice is generally smaller than a static slice. On the other hand, dynamic slice calculations require more memory and execution overhead for analysis of dynamic dependent relationships.

In this paper, the authors propose a new slice technology (the Call-Mark Slice) in order to resolve these problems. This technology has the following characteristics:

(1) Information resulting from a static analysis and small amounts of dynamic information on function calls is combined, and execution overhead is kept to a minimum level.

(2) The portion extracted as a slice is positioned between the static slice and the dynamic slice.

The authors installed their Call-Mark slice in a slice system [15] which had already been developed. In order to evaluate the validity of the Call-Mark slice, sample programs were executed on the system, and data were measured. The results showed that the Call-Mark slice could considerably limit the reference range compared to a static slice, as well as significantly reduce the load required to collect dynamic information during execution as compared to a dynamic slice.

Section 2 briefly describes static slices and dynamic slices. Section 3 explains the Call-Mark slice. In Section 4, the installation of the Call-Mark slice in the authors' system is described and examples of the execution of sample programs are given. Section 5 offers a comparison with other program slice extraction technologies. Section 6 provides a summary and suggests future topics.

2. Static Slices and Dynamic Slices

2.1. Static slices

Let us consider the statements s_1 and s_2 in a source program p. When the following conditions are fully satisfied, a control dependence (CD) from statement s_1 to statement s_2 is said to exist.

- Statement s_1 is a conditional statement.
- Whether or not statement s_2 is executed depends on the results of statement s_1.

This relationship is expressed as $s_1 \rightarrow s_2$.

When the following three conditions are all met, a data dependence (DD) is said to exist for the variable v from statement s_1 to statement s_2.

- The variable v is defined for statement s_1.
- The variable v is referenced for statement s_2.
- There is at least one path which can be executed from statement s_1 to statement s_2. In addition, there is no statement which defines the variable v between statement s_1 and statement s_2 for that path.

This relationship is represented as $s_1 \xrightarrow{v} s_2$.

The vertices on a program dependence graph (PDG) represent various statements, including inserted statements in the program, input and output statements, conditional judgment statements, and procedure statements. The edges represent the above two relationships among the statements. The program dependence graph for the Pascal source program in Fig. 1 is shown in Fig. 2.

The static slice [the pair (v, s) is referred to as the slicing base and represents the calculations for the slice for any statement] for the variable v in statement s in a program is the set of control dependence edges from the vertices corresponding to the slicing base and the statements corresponding to the vertices which can reach the data dependence edges by moving backwards.* For instance, when the static slice is calculated using the variable d in statement 24 of the program in Fig. 1 as a slicing basis, all statements except for the write statements (lines 12, 14, and 16) are included, as shown in Fig. 3.

2.2. Dynamic slices

In dynamic slice calculations, the dependence relationship is calculated for the source code, then the slices are

*When following the data dependence edges, first the edge for v is followed, then only the edge for the affected variable is followed using estimates.

```
 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
                                  ube: 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.
```

Fig. 1. Pascal source program.

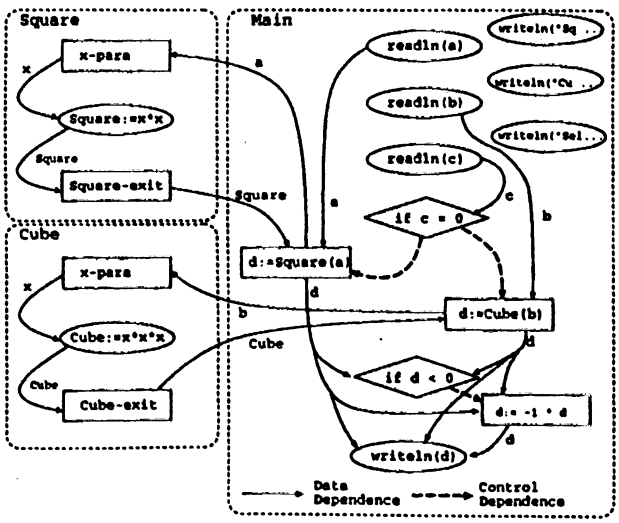

Fig. 2. Program dependence graph (PDG).

extracted. In dynamic slice calculations, the object of the calculations of the dependence relationship is an execution sequence. An execution sequence is a sequence of statements that is executed in practice when a particular input is given and a program is executed. Also, the execution of the p-th statement in an execution sequence is referred to as the execution time p.

Let us consider the execution times r_1 and r_2 in the execution sequence e. When the following conditions are all satisfied, dynamic control dependence (DCD) is said to exist from r_1 to r_2.

- The execution time r_1 is a conditional statement.
- Whether or not r_2 is executed depends on the results of r_1.

When the following conditions are all satisfied, dynamic data dependence is said to exist for the variable v from r_1 to r_2.

- The variable v is defined for r_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
13    readln(a);
14
15    readln(b);
16
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.
```

Fig. 3. Static slicing results by d at line 24.

- The variable v is referenced for r_2.
- There is no execution time which defines v between r_1 and r_2 for that path.

The slicing basis (x, r, v) for a dynamic slice consists of the input x, the execution time r, and the variable v. The dynamic slice corresponding to the slicing basis (x, r, v) is the set of statements which can be obtained by following in reverse the dynamic control dependence and the dynamic data dependence relationships from r.

Figure 4 shows the results of dynamic slicing calculations for the variable d, the execution time 13 (time at which the statement in line 24 is executed), and the input ($a = 2, b = 3, c = 0$) which is the slicing basis for the program in Fig. 1.

2.3. Features of dynamic slices and static slices

In the calculations for static slices, dependence analysis for the source program is performed without executing

```
 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
 8
 9
10
11  begin
12
13    readln(a);
14
15
16
17    readln(c);
18    if(c = 0) then
19       d := Square(a)
20
21
22
23
24    writeln(d)
25  end.
```

Fig. 4. Dynamic slicing result by d at line 24 with input ($a = 2, b = 3, c = 0$).

the program, and is implemented using the obtained PDG. When creating a PDG, the control dependence relationship can be readily found by checking the structure of the program. In addition, the data dependence relationship can be found by solving the data flow equation [3]. Although the complexity of the static slice calculations varies with the evaluation standards, in practice calculations can be performed in a short time [15, 17]. However, because a static slice considers all executable paths, in many instances, the calculation results for a static slice include a large portion of the original program, and so the reference range cannot be kept small.

On the other hand, because a dynamic slice is calculated from the execution sequence obtained by executing the program using particular input, parts unrelated to the execution may be included in the slice. In other words, in general the extracted size is smaller than a static slice. Let us in addition consider finding the causes of failures in a program when problems occur in a particular set of test data. In a static slice, calculations are performed up to the dependence relationship for the paths not executed for the test data, and so parts wholly unrelated to the cause of the fault may be included. However, because a dynamic slice calculates from the middle of a part related to the execution of the test data, parts unrelated to the fault will not be included in the slice. Thus, this is highly efficient when finding the location of faults in a particular set of test data.

Although analysis prior to program execution is not necessary for dynamic slice calculations, information about the dynamic dependence relationship and the dynamic control dependence relationship during execution must be stored in main memory. As a result, dynamic slice calculations require more memory and computational time. In addition, the execution sequence whose object is to extract a dynamic slice may, depending on the data, become extremely large because it is proportional to the number of statements the program executes. As a result, a considerable amount of extraction time may also be required.

3. The Call-Mark Slice

3.1. Outline

The Call-Mark slice proposed in this paper uses both static information and dynamic information in order to resolve the problems with static slices and dynamic slices. The problems with dynamic slices described in the previous section can be summarized as follows.

- Analysis of the execution sequence memory and the dynamic dependence relationship
- Slice calculations for the dynamic dependence relationship

Regarding the former problem, the authors decided to store only information on whether or not the function or procedural call statements are executed instead of precise information. As for the latter problem, the authors resolve it by analyzing the dynamic dependence relationship and not the static dependence relationship.

A Call-Mark slice is composed of a set of statements obtained by excluding statements for which the lack of effects can be calculated using the following methods from among those statements not affected by a static slice during execution of a program with particular input data. Because statements that are not affected are calculated, the authors' method focuses on whether or not a feature in a part of a program is executed, in other words, whether or not a function call statement is executed can be determined.

3.2. Definitions (ED, CED)

In order to calculate a Call-Mark slice, a new statement dependence relationship must be defined. An execution dependence (ED) is said to exist from statement s_1 to statement s_2 when the following condition is satisfied:

- Statement s_1 will not be executed when statement s_2 is not executed.

Although dynamic information is required in order to check all of the execution dependence relationships in a program, a subset can be found by using static analysis. In other words, when s_1 and s_2 are in the same basic block of the control flow graph (CFG) [3], that is, when there is no path that goes out of or enters into the two statements, s_1 and s_2 are in an execution dependence relationship. In addition, when s_1 is a basic block which controls the s_1 basic block, s_1 is execution dependent on s_2. Control flow and dominance relationships for the basic blocks can be readily found through static analysis [3].

Next we define the dominance call statement set $CED(s)$ with respect to statement s as follows:

$$CED(s) \equiv \{t \mid t \text{ is a function or a procedure call, and } s \text{ depends on the execution of } t\}$$

The execution of statement s is subordinate to the call statement included in $CED(s)$. When none of the call statements included in $CED(s)$ are executed, it can be determined that s has not been executed. The following simple program shows an example of *CED*.

```
      ...
s1: call A ;
s2: if a=1 then begin
s3:      b:= c ;
s4:      call B ;
   ...
```

Here, $s1$ and $s2$ depend on each other for their execution. $s3$ and $s4$ also depend on each other for their execution. Moreover, $s3$ depends on $s1$, and $s4$ on $s2$, for execution. As a result, $CED(s2) = \{s1\}$ and $CED(s3) = \{s1, s4\}$ can be obtained.

3.3. Input language

In order to compare static slices and dynamic slices in this paper, the authors' method was used in a slice system previously developed [15]. As a result, the input language used in this slice system was also used as the input language for the Call-Mark slice. The input language is as follows. This language has conditional statements (if statements), substitution statements, repeat statements (while statements), input statements (readln statements), output statements (writeln statements), procedure call statements, and compound statements (begin-end statements) as statements. Only scalars will be considered as a variable type. A program consists of global variable declarations, procedure (function) definitions, and a main program, and does not include a block structure. Only local variables and virtual argument variables defined in a declaration within a procedure, and global variables, can be referenced. Local variables within other procedures cannot be referenced. Procedures can be defined to be self-recursive or mutually recursive, and their arguments are treated as being passed by value.

Although the above language is used as an input language in this paper, execution-dependent relationships can be calculated even for languages which include non-structural statements such as pointer variables and goto statements. As a result, the Call-Mark slice can be applied to this kind of language.

3.4. Definition of the Call-Mark slice

First, an intuitive definition of the Call-Mark slice will be given. For the execution e with respect to the input x in a program P, the set S_1 of statements for which the entire dominant call statement set in the program is executed will be considered. When the static slice with respect to the slicing basis (s, v) for P is S_2, $S_1 \cap S_2$ can be referred to as "the Call-Mark slice for the slicing basis (x, s, v)." This means that statements which are included in the Call-Mark slice but not in the static slice are either not executed in e or have an execution-dependent relationship with the statements that are not executed.

The Call-Mark slice can be defined formally by using Steps 1 through 3.

[Step 1] Analysis of dependency relationships prior to execution

Inputs
PDG: Program Dependence Graph
CM: Set of executed call statements.
(s_c, v): Slicing bias, s_c is a statement, v is a variable name.
Temporary
M, N: Set of nodes
m, n: Nodes
Output
M: The set of nodes obtained as the Call-Mark slice.
Algorithm Body
(1) $M \leftarrow s_c$
(2) $N \leftarrow \{n \mid n \xrightarrow{v} s_c\} \cup \{m \mid m - - \rightarrow s_c\}$
(3) While $N \neq \phi$ The following is repeated.
  (a) One of $n \in N$ is selected
  (b) $N \leftarrow N - n$
  (c) if $CED(n) \not\subseteq CM$, return to (a)
  (d) $M \leftarrow M \cup n$
  (e) $N \leftarrow N \cup \{m \mid m \notin M \wedge (m \xrightarrow{w} n \vee m - - \rightarrow n)\}$
    Here, w is the name of each variable referenced at statement n

Fig. 5. Algorithm of postexecution collection for Call-Mark slicing.

```
 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
 8
 9
10
11  begin
12
13    readln(a);
14
15
16
17    readln(c);
18    if(c = 0) then
19       d := Square(a)
20
21
22    if (d < 0) then
23       d := -1 * d;
24    writeln(d)
25  end.
```

Fig. 6. Call-Mark slicing result by d at line 24 with input ($a = 2$, $b = 3$, $c = 0$).

As with the calculations for the static slice, the data dependence relationships and control dependence relationships are analyzed, and the PDG is created.

[Step 2] Recording during execution

The program is executed, and for each function or procedure call statement that is executed, "executed" is recorded in the PDG node corresponding to the call statement. The set of recorded call statements is referred to as CM.

[Step 3] Extraction of the Call-Mark slice

The Call-Mark slice is extracted from the algorithm in Fig. 5.

Figure 3 shows the static slice for the slicing basis (line 24, d) in the program in Fig. 1. Let us now consider the execution of this program using the input ($a = 2$, $b = 3$, $c = 0$). In this case, $CM = \{19\}$, and Fig. 6 represents the Call-Mark slice for a slicing basis similar to the static slice.

4. Using the Call-Mark Slice

4.1. Outline of the slice system

In order to evaluate the Call-Mark slice, the authors added the Call-Mark slice function to the slice system [15] using a Pascal program developed by the authors' group (Fig. 7). In this system, the source program is analyzed into

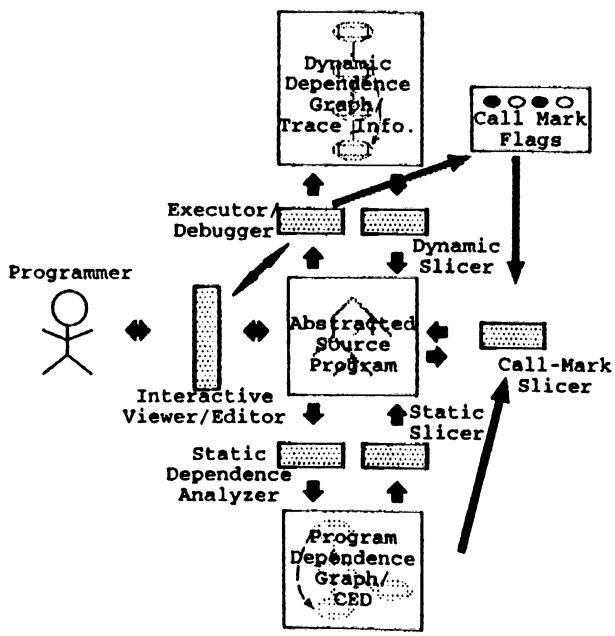

Fig. 7. Architecture of slicing system.

an abstracted source program and then recorded. The user can interactively reference or alter the source program via a visual editor. The source program is converted to a PDG analyzed via user input. The static slice is calculated in the PDG using the designated slicing standard. The source program and the static slice can be executed via the interpreter (executor). The debugger has functions to set tracing and break points. The dependence relationships for dynamic variables can be recorded during execution and then used to calculate the dynamic slice. The overall size of the system is roughly 19,000 lines in C language, including the portions related to the Call-Mark slice. The Call-Mark slice was put in using the method described in the previous section.

4.2. Program execution

Using this system the authors executed several programs. Table 1 shows the measured data for three programs (P1: calendar output; P2: bar problem [20], P3: expanded bar problem). Although these values differed depending on the slicing basis and input data, in the current experiment canonical debugging conditions were conceived, and so these were selected (in many instances, the slicing basis is an output variable that comes after a program is finished).

Table 2 shows the analysis time required before execution. This represents the time necessary for PDG structuring for the static slice and the time to calculate the CED and the PDG structure for the Call-Mark slice. No analysis was necessary for a dynamic slice at this stage.

Table 3 shows the time needed to execute the program. Although not necessary to calculate the static slice, this was recorded for purposes of comparison with the time required to execute the original program. When executing the dynamic slice, this time is also included because the analysis of dynamic dependence relationships between variables is performed at the same time. In the Call-Mark slice, the time required to record the function call statements is included in the execution time.

Table 4 shows the time required to collect the slices. For a static slice, this is the time required to calculate the dependence relationships in the program dependence graph. For a dynamic slice, this is the time required to calculate the dynamic dependence relationships. For the Call-Mark slice, this is the time required in Step 3.

Table 1. Size of various slicing results (lines of code)

program	static	dynamic	Call-Mark
P1 (88 lines)	27	14	22
P2 (387 lines)	175	139	156
P3 (941 lines)	324	50	166

Table 2. Preexecution analysis time (ms)

(Pentium-II 300 MHz with 256-MB Memory)

program	static	dynamic	Call-Mark
P1	22	N/A	23
P2	1,275	N/A	1,362
P3	5,652	N/A	8,670

Table 3. Execution time (ms)

(Pentium-II 300 MHz with 256-MB Memory)

program	static	dynamic	Call-Mark
P1	38	87	47
P2	48	903	53
P3	4,046	31,635	4,104

Table 4. Slice collection time (ms)

(Pentium-II 300 MHz with 256-MB Memory)

program	static	dynamic	Call-Mark
P1	1	199	1
P2	5	2,863	8
P3	93	1,182	80

5. Discussion

5.1. Program execution time

- Extracted slice size

As shown in Table 1, the size of the Call-Mark slice is a value between the size of the static slice and the dynamic slice.

The Call-Mark slice extracts statements that fulfill the following conditions from the static slice.

(Condition 1) Statements that can be determined not to have been executed based on CM and CED

(Condition 2) Only statements that are determined not to have been executed and statements that have a dependence relationship

Consequently, the Call-Mark slice is a subset of the static slice with respect to the same slicing basis.

The dynamic slice does not include statements which satisfy the above two conditions. In addition, although the Call-Mark slice is not actually executed, even if CED and CM are used, there will be statements that cannot be excluded. For instance, if a particular statement *S* is present

in a block whose execution is subordinate to a particular conditional statement, and if there is no function or procedure call statement in that block or the upper-level block over it, then because $CED(S) = \phi$, it cannot be determined whether or not that block is executed, and that block ends up being included in the Call-Mark slice. Therefore, the dynamic slice is a subset of the Call-Mark slice.

- Preexecution analysis time

As shown in Table 2, a little extra time is required for the Call-Mark slice compared to the static slice. This is because analysis of the execution dependence relationships is necessary in addition to the PDG structure analysis.

- Execution time

As shown in Table 3, there is considerable overhead in the dynamic slice. In the execution of *P*3, 90 MB of real memory is used. If the execution of this program were longer, this overhead would increase that much more. On the other hand, in the Call-Mark slice, there is little rise in the overhead compared to the execution (execution of the original program) in the calculation of the static slice. This indicates that the time required to record the call statements is extremely brief.

- Slice collection time

As shown in Table 4, a long time is needed to collect the dynamic slices. The Call-Mark slice time is roughly the same as the static slice time. This is also smaller than the time required for the dynamic slice in program *P*3. This is because the PDG range is smaller than the static slice so as to exclude the parts from the PDG that are unrelated to execution.

5.2. Related research

The Call-Mark slice is an efficient, practical method to limit the reference range for a programmer. The preexecution analysis time, the execution time, and the time required for slice collection are all roughly the same as for a static slice, and the validity of the slice is greater than that for a static slice; thus, this method can be thought of as a good balance between effectiveness and validity. Recently research other than the Call-Mark slice has also been performed on combining static information and dynamic information [4, 6].

In Ref. 1, a method for finding a slice from a static dependence graph is described as an approach to collecting slices using dynamic information. However, the method described in this paper requires information regarding whether or not each statement is executed, and so execution overhead and a larger storage region are required in order to store the information compared to the Call-Mark slice. In addition, although the information for each statement must be stored during execution for the slices calculated using this approach, the result is that extra statements are included in the slice compared to a dynamic slice [12].

A hybrid slice [12] improves a static slice by using breakpoints and function calling history information. The former is given by the programmer, and is then used to estimate the executed control flow. The latter is used to calculate the dynamic slice between function calls and the return point. Because this method uses more dynamic information than the authors' method, it is closer to being a dynamic slice than the authors' method, but breakpoints must be set at appropriate locations in order to improve the slicing results. On the other hand, the authors' method can provide input data and perform calculations automatically except for designating the slicing basis. In addition, a hybrid slice requires a very large storage region in order to store the calling history. The size required depends on the length of the execution sequence. In the authors' method, however, the only region necessary is for storing the execution of call statements, which is proportional to the size of the program. When the program execution is long, the difference in the necessary storage region becomes evident.

A constrained slice [4] is a generalization of a static slice and a dynamic slice. It uses a method to execute the coding of a program. Constraint conditions are given as input. Using this input constraint, a program is rewritten, and then the dependence relationships are analyzed. This method includes a generalization of static slices and dynamic slices, a partial evaluation, and a simplification of the program. Though interesting, it is not known if this generalized method can be used efficiently or effectively.

6. Conclusions

Restricting the range to be considered by a programmer is extremely important for improving efficiency in debugging and maintenance. Conventional program slicing methods have problems with efficiency and effectiveness. Thus, the authors proposed a Call-Mark slice as an efficient and effective program slicing technology. This technology can analyze and execute dependence relationships in roughly the same time as for a static slice. In addition, the size of the slice obtained is smaller than a static slice and larger than a dynamic slice. The authors created a slice collection algorithm, then evaluated their technology.

In the present system, static dependence analysis (PDG creation) was performed prior to executing the program. In the future the authors will study static dependence analysis after program execution (after obtaining Call-

Mark information). This should allow for the elimination of dependence relationship calculations for parts unrelated to execution and thereby further reduce the dependence analysis time.

REFERENCES

1. Agrawal H, Horgan J. Dynamic program slicing. SIGPLAN Notices 1990;25:246–256.
2. Agrawal H, Demillo RA, Spafford EH. Debugging with dynamic slicing and backtracking. Software Practice and Experience 1993;23:589–616.
3. Aho AV, Sethi R, Ullman JD. Compilers: Principles, techniques, and tools. Addison–Wesley; 1986.
4. Field J, Ramalingam G. Parametric program slicing. Proc 22nd ACM Symposium on Principles of Programming Languages, p 379–392, San Francisco, 1995.
5. Atkinson DC, Griswold WG. The design of whole-program analysis tools. Proc 18th Int Conference on Software Engineering, p 16–27, Berlin, 1996.
6. Gupta R, Soffa ML. Hybrid slicing: An approach for refining static slices using dynamic information. Proc 3rd Int Symposium on the Foundation of Software Engineering, p 29–40, 1995.
7. Harrold MJ, Ci N. Reuse-driven interprocedural slicing. Proc 20th Int Conference on Software Engineering, p 74–83, 1998.
8. Horwitz S, Reps T. The use of program dependence graphs in software engineering. Proc 14th Int Conference on Software Engineering, p 392–411, 1992.
9. Korel B, Laski J. Dynamic program slicing. Inf Process Lett 1988;29:155–163.
10. Korel B, Laski J. Dynamic slicing of computer programs. J Syst Software 1990;13:187–195.
11. Murphy GC, Notkin D. Lightweight lexical source model extraction. ACM Trans Software Eng Methodol 1996;5:262–292.
12. Naoi K, Takahashi N. Program slicing using a path dependence flow graph. Trans IEICE 1995;J78-D-I:607–621.
13. Ning JQ, Engberts A, Kozaczynski WV. Automated support for legacy code understanding. Commun ACM 1994;37:50–57.
14. Nishimatsu K, Kusumoto S, Inoue K. Experimental evaluation of a program slice in a fault location finder. Tech Rep IEICE 1998;SS98-3.
15. Sato S, Iida H, Inoue K. Test of a debugging support system based on program dependence relationship analysis. Trans Inf Process Soc 1996;37:536–545.
16. Shimomura T. Program slicing technology and applications. Kyoritsu Publishing; 1995.
17. Ueda R, Lin L, Inoue K, Shimai H. A method for calculating slices in a program which includes recursion. Trans IEICE 1995;J78-D-I:11–22.
18. Vengatesh GA, Fischer CN. SPARE: A development environment for program analysis algorithms. IEEE Trans Software Eng 1992;18:304–315.
19. Weiser M. Program slicing. Proc 5th Int Conference on Software Engineering, p 439–449, 1981.
20. Yamazaki T. An explanation of a program design method using common problems. Inf Proc 1984;25:934.

AUTHORS (from left to right)

Akira Nishimatsu graduated from the Department of Informatics of Osaka University in 1997 and completed the M.E. program in 1999. Currently he is affiliated with NTT Data COE System Headquarters. He is pursuing research on program slicing.

Minoru Jihira graduated from the Department of Informatics of Osaka University in 1998. Currently he is in the M.E. program at Nara Institute of Science and Technology. He is pursuing research related to program slicing and user interfaces.

Shinji Kusumoto (member) graduated from the Department of Informatics of Osaka University in 1988. While in the doctoral program, he became a lecturer in the Department of Informatics in 1991, and has been an instructor since 1996. He holds a D.Eng. degree. He is pursuing research related to project management and quantitative evaluations of manufacturing and quality for software. He is a member of the Information Processing Council and IEEE.

Katsuro Inoue (member) graduated from the Department of Informatics of Osaka University in 1979 and completed the doctoral program in 1984. From 1984 to 1986 he was an assistant professor at the University of Hawaii at Manoa. In 1989 he became an instructor in the Department of Informatics of the School of Engineering Science, Osaka University, an associate professor in 1991, and a professor in 1995. He holds a D.Eng. degree. He is pursuing research related to software engineering.