Empirical Software Engineering, 7, 49–76, 2002.

Experimental Evaluation of Program Slicing for Fault Localization

SHINJI KUSUMOTO kusumoto@ics.es.osaka-u.ac.jp
AKIRA NISHIMATSU
KEISUKE NISHIE inoue@ics.es.osaka-u.ac.jp
KATSURO INOUE
Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University, 1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan

Abstract. Debugging large and complex software systems requires significant effort since it is very difficult to localize and identify faults. Program slicing has been proposed to efficiently localize faults in the program. Despite the fact that a number of debug systems using program slicing, have been developed, the usefulness of this method to fault localization has not been sufficiently evaluated. This paper aims to experimentally evaluate the usefulness of the program slicing method to fault localization. In order to conduct the experiment, we first developed a debug tool based on program slicing, after which two experimental projects were conducted, in which subjects (debuggers) were divided into two groups. A program that includes several faults is given to each subject of the group. Each subject in Group 1 localizes the faults by using the slicing-based method, whereas in Group 2 each subject localizes the faults by using the conventional debugger-based method. Finally, the effectiveness of program slicing is analyzed by comparing the data collected from both groups. As the results of these experiments, we confirm that the program slicing method is indeed useful to localize program faults.

Keywords: Program slice, empirical evaluation, fault localization, tool, measurement.

1. Introduction

Increasing the productivity and quality of software development processes is an important research objective in software engineering. In order to improve the productivity and quality, the software development processes have to be improved (Frakes, et al., 1991; Humphrey, 1989).

It is widely recognized that errors have a large impact on software productivity and quality. In attempting to reduce the number of delivered faults, it is estimated that most companies spend between 50 and 80% of their development effort on test process (Collofello and Woodfield, 1989). Therefore, reducing the effort of test process is an important step towards increasing productivity and quality in the software development processes.

The test process would consist of two sub-processes: (1) testing, i.e., detecting whether failures[1] happen in the target program execution, and (2) debugging, i.e., localizing the faults[2] that are the causes of the failures and correcting the program. According to Myers (1979), the fault-localizing activity in the debugging sub-process

needs 95% of the debugging effort. Thus, it is very important to develop an effective fault-localizing method.

Program slicing has been proposed (Agrawal and Horgan, 1990; Korel and Laski, 1990; Weiser, 1982, 1984) and applied to localize faults in the program (Agrawal et al., 1993; Fritzson et al., 1991; Lyle and Weiser, 1987). By definition slicing is a technique which extracts all statements that affect some set of variables in the program. The set of all extracted statements is called a slice. In the worst case, the extracted slice may be the whole program and in such a case very much effort is needed to localize the fault in the slice. However, it has been experimentally confirmed that in realistic applications the extracted slice was reasonably small compared to the entire program (Venkatesh, 1995).

On the other hand, until now, numerous studies in software engineering have developed new methods, tools, or techniques to improve some aspect of software development or maintenance. However, relatively little evidence has been gathered on which of these new developments are effective (Tichy et al., 1993). For example, traditional reading techniques have been empirically evaluated for several years (Basili, 1997). In the area of program slicing, the usefulness of the slicing method to fault localization has not been sufficiently evaluated.

The present study aims to experimentally evaluate the potential usefulness of program slicing to fault localization. The first stage in this study was to construct a software debugging support tool based on the slicing method. This tool includes the fundamental debugging functions and the extracting program slice from the target program. By using the tool, we then conducted two experimental projects to evaluate the usefulness of the slicing for fault localization. In both experiments subjects (debuggers) are divided into two groups. Next, a number of programs, each of which included several faults, were given to each subject in the group. Each subject in the first group was given the task to localize the faults by using the slicing-based method. On the other hand, each subject in the second group was asked to localize the faults by using the conventional debugger-based method. Finally, we analyze the effectiveness of the program slice by comparing the data collected from both groups. As will be shown, based on these experiments, we conclude that the program slice is useful to localize the faults.

Section 2 describes the definition of the program slicing method and presents a literature survey about application of this method to fault localization. Next, Section 3 describes the debugging support tool which was developed for these experiments. Sections 4 and 5 describe the experimental projects and analyze the experimental results. In Section 6 the validity and limitation of our experiments are discussed, as well as future research topics. Finally, Section 7 concludes this paper.

2. Preliminaries

2.1. Software Debugging

According to IEEE, Standard Glossary of Software Engineering Terminology (1990), the test phase is defined to be the process of exercising or evaluating a system

or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results. The test phase includes several kinds of testing. Unit testing, integration testing and acceptance testing are the better-known ones. Unit testing is the verification of a single program module which may include several functions or procedures, in an isolated environment (i.e., isolated from all other modules) to see whether the unit satisfies the specified unit design specification. Integration testing is the verification of the interfaces among modules to see if the system behaves as the specified design specification. Acceptance testing is the validation of the system or program to the requirements specification.

If failures are detected during testing, the faults, which cause the failures, must be corrected. This activity is called debugging. Debugging can be considered as three successive activities: (1) failure localization, (2) fault localization and (3) fault correction. During failure localization, the failures which have been detected in the testing are reproduced. During fault localization, location of the fault in the target program is identified. Then, during fault correction, the fault is corrected. Finally the modified program is re-tested.

According to Myers (1979), fault localization takes most of the debugging effort and it is necessary to develop effective fault-localizing methods.

2.2. Program Slice and its Application to Debug Process

Program slicing has been proposed (Agrawal and Horgan, 1995; Korel and Laski, 1990; Weiser, 1982, 1984) and applied to localize faults in the program (Agrawal et al., 1993; Fritzson et al., 1991; Lyle and Weiser, 1987). By its nature, slicing is a technique which extracts all statements that affect some set of variables in the program. The set of all extracted statements is called a slice.

Slicing techniques have been usually divided into static and dynamic slicing. A program slice for a variable x at a statement n in program P ((x,n) is called slicing criterion), by definition means a set of program statements and/or predicate expressions in P, which possibly affect the value of x at n. The analysis result for all possible input data is called static slice (Weiser, 1984), and that for a particular input instance is called dynamic slice (Korel and Laski, 1990).

The outline of the slicing-based debugging is as follows: At first, the slice for variables, which have incorrect values at some statement for program execution, is calculated. Then, a set of statements that include the faults in the slice is identified.

In the worst case, the extracted slice may be the whole program and in such a case the effort needed to localize the fault in the slice becomes too extensive. However, it has been experimentally found that the extracted slice for real applications was reasonably small compared to the entire program (Venkatesh, 1995).

In order to support the slicing-based debugging, several debug-support systems using slicing techniques are available. For example, program error-locating assistant system (PELAS) can guide the programmer during the error localization process for

Pascal programs (Korel, 1988). PELAS makes use of the dependence network to reason about probable sources of the error. It is based on the idea that any instruction in the execution trace, from the beginning to a position of incorrectness, could conceivably have been responsible for the faulty behavior. However, the usefulness of the slicing method and of related systems to fault localization has not been sufficiently evaluated, and the present study was aimed to empirically address this issue.

3. Debug Supporting Tool for Experimental Evaluation

In this Section, we describe the debug supporting tool used for our experiments.

3.1. Overview of the Tool

We developed the debug supporting tool according to the following concepts:

(P1) The target program of the tool should be a well-known programming language and could be applicable to the practical software development.

(P2) The tool should calculate the program slice efficiently and also contain the fundamental functions which conventional debuggers (e.g, dbx) have.

(P3) The tool should provide easily-handled user-interface.

With respect to (P1), we use a subset of the Pascal language which has the following features: (1) Conditional statement (`if`), assignment statement, iterative statement (`while`), input statement (`readln`), output statement (`writeln`), procedure-call statement, and compound statement (enclosed by `begin`–`end`) are available. (2) Procedures can be defined self- and mutual-recursively, with arguments of either call-by-value or call-by-reference. (3) All the variables in the program are standard type (Boolean, character, integer and real type). Also array, the element of which is standard type is available.

Although this language seems to be simple, it contains most of the fundamental structures and statements of procedural languages, including recursive procedure calls, but no pointer types.

With respect to (P2), we implement the function of calculating the static slice. Most of the conventional slicing system uses the dynamic slice that is calculated from the program execution trace. However, the use of dynamic slice has the following two major problems:

- The system requires a large amount of memory to keep the execution trace and its analysis results.
- Dynamic slice must be always recalculated after every program execution.

In our experiments, the performance of the tool is very important, because if the calculation of the slice needs much time, the subject could become irritated and lose the sense of purpose. Thus, we adopted the static slice in the experiment. Also, our tool includes the execution step, referring the values of variables, setting the breakpoints, etc. (That is, these functions are the same as the conventional debugger).

With respect to (P3), the usability of the tool is the most important factor in our experiments. Lyle and Weiser (1987) attempted to evaluate program slicing as a debugging tool and found that subjects using the tool debugged slower than subjects who were not using the tool. They concluded that the slower performance was caused because (1) the subjects tended to use the tool in an inappropriate manner and (2) the presentation of the slice to the users might have led to confusion. Thus, we designed our tool in such a way that it should provide the graphical user interface to the user and the presentation of the slice should be intuitively understandable. Using the interface, the subject can easily debug the program and calculate the slice.

3.2. Functions and System Architecture

The system has the following functions (F1) and (F2):

(F1) Step execution, referring the values of variables, setting the breakpoints, etc.

(F2) Extracting the program slice (both backward slice and forward slice).

The system architecture is shown in Figure 1. Briefly, this system is composed of editor, analyzer, interpreter, debugger and slicer. The editor is used to edit the programs. The analyzer performs syntax analysis and creates program dependence graph (PDG). The slicer calculates program slices using PDG. The details of slice calculation are shown in the Appendix and by Sato et al. (1996). The interpreter executes either the original program or the sliced/transformed program, and the debugger provides several functions (the step execution, referring the values of variables, setting the breakpoints, etc.) to debug the programs.

A user at first enters a target program to the system using the editor and then analyzer generates the PDG and other information (such as the syntax tree and variable information).

Next, the user would specify various operations to the system. If program execution is specified, the interpreter gets the information from the analyzer, then executes the program. The interpreter also features many useful operations normally provided by the debugger, such as step-by-step execution, break point setting, and variable value inspection.

If some wrong output is found during an execution, the user specifies an operation to extract only the statements which may have had caused the wrong output. In such a case the user may use the slicing operation to narrow the range where the faults

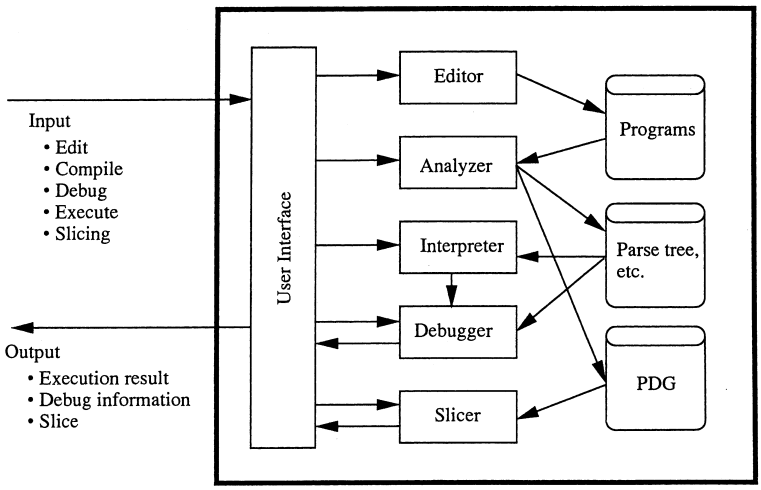

Figure 1. System architecture.

would exist. If program slicing is selected, the slicer reads PDG from the analyzer and also reads the slicing criterion (i.e. output variable to be focused), after which the slice is calculated. The result is marked on the original program code.

3.3. Execution Example

Figure 2 shows the main screen of the system. It includes the following three windows:

(1) Editor window: The target program can be constructed and modified through this window. The result of slicing is also shown.

(2) Status window: It shows the status of the system execution.

(3) Display window: It shows the result of program execution.

As an example of a debugging process with our system, we used a faulty program in Figure 3. This program is intended to calculate the summation, maximum, minimum, and average value of five integers given by the user. The designed tool is used as follows: The program is entered into our system, and it is executed using the interpreter. It can be observed that the `MAX` value is incorrect. Then, the backward slice is extracted with this value (variable `Max` at line 32). The slicing criterion is specified in the window. In this example, we specify a variable `Max` at line 32 as a slicing criterion. Statements displayed with an emphasis belong to the result slice.

By inspecting the slice result, we can find that the statement line 14 directly affects the value of `Max` in the statement line 32.

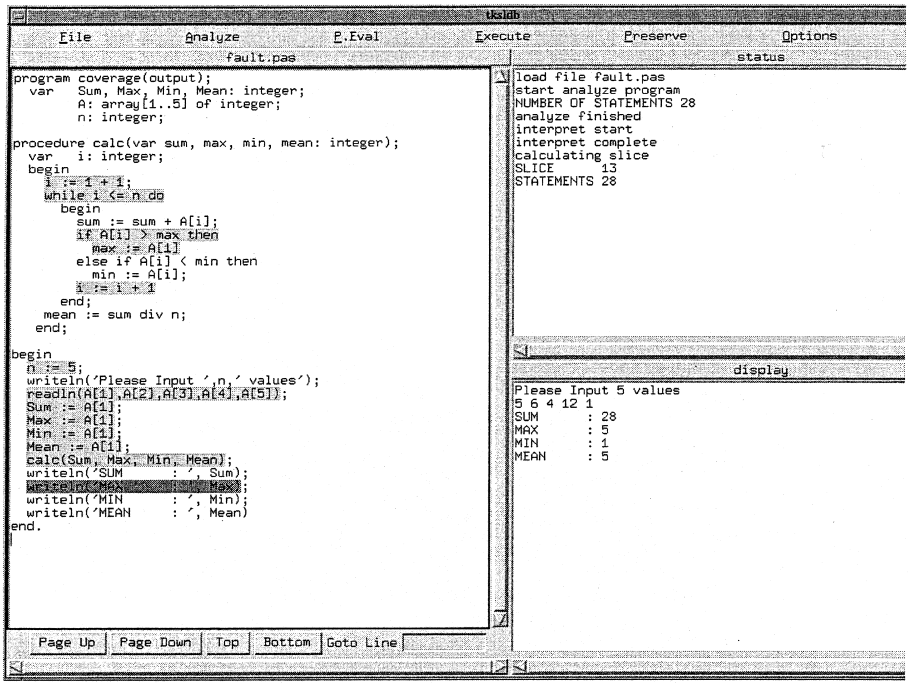

Figure 2. Main screen of the system.

Now, we set a break point at statement line 14, and execute the slice result as a program. During the execution, we find that statement line 14 is executed many times but the value of `Max` does not change at all. This is because the variable referred here is `A[l]`, instead of `A[i]`. By correcting it to `A[i]`, we finally get the right output value.

4. First Experiment

Here, we describe the controlled experimental projects to evaluate the usefulness of the program slicing for faults localization. In these experiments we compare two fault localization methods: the slicing-based method and the conventional debugger-based method. The slicing-based method is to use all the functions of our tool in Section 3, whereas the conventional debugger-based method is to use only the conventional-debugger function of it.

4.1. Design of Experiment 1

4.1.1. Goal

The overall goal of Experiment 1 was to examine whether program slicing can improve individual fault localization compared to conventional debugger-based fault

```
 1  program coverage(input,output);
 2    var   Sum, Max, Min, Mean: integer;
 3          A: array[1..5] of integer;
 4          n: integer;
 5
 6  procedure calc(var sum, max, min, mean: integer);
 7    var   i: integer;
 8    begin
 9      i := 1 + 1;
10      while i <= n do
11        begin
12          sum := sum + A[i];
13          if A[i] > max then
14            max := A[1]
15          else if A[i] < min then
16            min := A[i];
17          i := i + 1
18        end;
19      mean := sum div n;
20     end;
21
22  begin
23    n := 5;
24    writeln('Please Input ',n,' values');
25    readln(A[1],A[2],A[3],A[4],A[5]);
26    Sum := A[1];
27    Max := A[1];
28    Min := A[1];
29    Mean := A[1];
30    calc(Sum, Max, Min, Mean);
31    writeln('SUM      : ', Sum);
32    writeln('MAX      : ', Max);
33    writeln('MIN      : ', Min);
34    writeln('MEAN     : ', Mean)
35  end.
```

Figure 3. Example target program including a fault.

localization. To document our goal, we used the goal/question/metric (GQM) goal template of the GQM approach (Basili et al. 1994): Analyze slicing and conventional debugger-based fault localization for the purpose of their evaluation with respect to their fault localization effectiveness from the view point of the researcher in the context of the experimental projects at the Osaka University.

4.1.2. Variables

In Experiment 1, we manipulated two independent variables.

1. The fault localization technique used by a subject (slicing-based and conventional debugger-based).
2. The target program to localize the faults (two programs developed independently from the same requirements specification).

For each fault localization, we measured one dependent variable.

1. The individual fault localization effectiveness defined as the time needed to localize the faults included in the program.

4.1.3. Design

The design of Experiment 1 is shown in Table 1. As discussed above, the subjects applied two different fault-localizing techniques (slicing-based and conventional debugger-based) to two programs. The subjects were assigned randomly to the two groups. Two sub-experiments were conducted(Trial-1 and Trial-2).

4.1.4. Hypothesis

We investigated the following hypothesis (H1): There is a difference between subjects using slicing-based fault localization and subjects using conventional debugger-based fault localization with respect to their mean fault localization effectiveness.

Table 1. Design of Experiment 1.

	Group 1	Group 2
Trial-1	PG1/Slicing-based fault localization	PG1/Conventional debugger-based fault localization
Trial-2	PG2/Conventional debugger-based fault localization	PG2/Slicing-based fault localization

4.1.5. Subjects

The six subjects of Experiment 1 were senior-graduate students of the Faculty of Engineering Science at Osaka University enrolled in the Software Engineering Laboratory. They have learned the fundamental software engineering principles through lectures and practical exercises. Also, they have experience in developing compilers for the subset of the Pascal language, which is also the language of the input program to our debug supporting tool shown in Section 3. Thus, they have thorough knowledge of the language. Also, the functions of the system, which each group had to use, were explained to the subjects prior to Experiment 1 being carried out.

They were divided into two groups: G1 and G2. Students A1, A2 and A3 were in G1, and students B1, B2 and B3 were in G2.

In Trial-1, subjects in G1 use the functions (F1) and (F2) in the debug supporting tool and ones in G2 use only the function (F1) in the debug supporting tool.

In Trial-2, subjects in G2 use the functions (F1) and (F2) in the debug supporting tool and those in G1 use only the function (F1).

4.2. Programs Used in Experiment 1

In Experiment 1, we used two programs (PG1 and PG2) which were developed based on the same specifications as for the inventory control program at a wine shop (Lian et al., 1997). However, since they were developed independently, their data structures and algorithms were not identical. The following eight kinds of faults were introduced into the program PG1.

($f1_1$)Lack of the output processing,

($f1_2$)Illegal assignment,

($f1_3$)Illegal conditional statement,

($f1_4$)Omission of the initialization,

($f1_5$)Lack of the procedure call,

($f1_6$) Wrong data renovation,

($f1_7$)Wrong parameter for the procedure call, and

($f1_8$)Wrong execution order for some procedure calls.

The following nine kinds of faults were introduced into the program PG2.

($f2_1$)Illegal conditional statement,

($f2_2$) Illegal conditional statement,

($f2_3$) Wrong reference to array variable,

($f2_4$) Wrong execution order for some procedure calls,

($f2_5$) Wrong parameter for the procedure call,

($f2_6$) Lack of the procedure call,

($f2_7$) Wrong data renovation,

($f2_8$) Illegal output, and

($f2_9$) Wrong registration for database.

The faults were actually introduced by several students in Osaka University. In the programming seminar, we had used the inventory control program at wine shop for several years, and had collected the faults introduced by the students. For example, ($f1_3$) and ($f1_6$) are shown in Figure 4.

For Trial-1, based on the PG1, we constructed a series of eight programs, called $PG1_i (i = 1, 2, \ldots, 8)$, by using the faults $(f1_1) \ldots (f1_8)$. $PG1_j$ includes faults $(f1_j)$, $(f1_{j+1}), \ldots, (f1_8)$. For example, $PG1_1$ includes $(f1_1), (f1_2), \ldots, (f1_8)$ and $PG1_5$ includes $(f1_5)$, $(f1_6), \ldots, (f1_8)$, respectively.

```
...
 while (IraiHyouVflag[i]=1) do
  begin                                                    (j<i)
    j:=1;
    ZaikoHikaeFlag:=0;
    while (ZaikoHikaeFlag=0)and(j<=i) do
    begin
      if IraiMeigara[j] = IraiMeigara[i] then ZaikoHikaeFlag := 1;
      j:=j+1
    end;
...
```

(*f13*)

```
...
if ZaikoHikaeFlag=0 then
    begin
      SoukoKensaku(IraiMeigara[i],IraiRyou[i]);
      if SyukkoFlag=1 then
      begin
        SakujyoSakeDB(IraiMeigara[i],IraiRyou[i]);
        SyukkoSiziHyou(i)
      end                                          SyukkoSiziHyou(i);
    end;                                           KousinIraihyouDB(i)
i:=i+1
...
```

(*f16*)

Figure 4. Examples of faults.

These faults correspond to the fundamental functions which can be identified by the subjects from the requirements specifications, the subjects can easily find the failure caused by each fault in the program execution. Also, eight test cases were prepared (testcase $PG1_1, \ldots,$ testcase $PG1_8$), each of which detects each corresponding fault $(f1_1) \ldots (f1_8)$. Moreover, in order to measure the time for localizing each fault, testcase $PG1_i$ is designed such that it is effective to detect only the fault $(f1_i)$ and other faults are masked not to be detected by the testcase $PG1_i$.

Similarly, for trial-2, we constructed a series of nine programs, called $PG2_i (i = 1, 2, \ldots, 9)$, by using the faults $(f2_1) \ldots (f2_9)$. Also, we prepared nine test cases (testcase $PG2_1, \ldots,$ testcase $PG2_9$), by which we can detect the faults respectively in order of the number of them.

4.3. Process of Experiment 1

Trial-1 is executed as the following process:

Step 0: $i = 1$.

Step 1: By using testdata $PG1_i$, the subject S localize the fault in program $PG1_i$.

Step 2: S reports the fault and the location to the supervisor. If the report is correct, then S goes to Step 3. If it is wrong, then S goes back Step 1.

Step 3: If $i = 9$ then, the experiment is completed. If $i < 9$ then $i = i + 1$ and we go back to Step 1.

As described above, in $PG1_i$, the only fault that can be detected by testdata $PG1_i$ is $f1_i$. In the next program, $PG1_{i+1}$, the fault has been fixed. Thus, the only part which is different between $PG1_i$ and $PG1_{i+1}$ is the one modified to fix the fault $f1_i$.

The data collected by the experiment supervisor are the time it took to localize the fault and the way this operation was performed. With respect to the subjects who use the slicing-based method, the data collected refers to how the slicing tool is used.

Trial-2 was performed in the same way.

4.4. Result and Analysis

4.4.1. Size of the Slice

Tables 2 and 3 show the size of the program in Experiment 1 and of the slice obtained by the typical slicing criterion for failure of each program. Also, the tables include the distance (number of lines) from the statement at which the incorrect values are printed, to that at which the fault is localized. With respect to the $PG2_9$, there were faults in two statements.

Table 2. Size of program and slice in Trial-1.

	$PG1_1$	$PG1_2$	$PG1_3$	$PG1_4$	$PG1_5$	$PG1_6$	$PG1_7$	$PG1_8$	Average
LOC	427	432	428	428	428	429	434	434	430
Size of slice	194	194	196	196	199	199	202	211	199
Distance	110	111	120	5	131	21	47	16	–

Table 3. Size of program and slice in Trial-2.

	$PG2_1$	$PG2_2$	$PG2_3$	$PG2_4$	$PG2_5$	$PG2_6$	$PG2_7$	$PG2_8$	$PG2_9$	Average
LOC	376	378	378	378	378	378	379	390	391	381
Size of slice	161	159	160	160	161	160	162	165	166	162
Distance	124	52	100	24	65	167	116	9	110/8	–

From these tables, the average ratios of the slice to the entire program are 46 and 42%, respectively. The extracted slice in each trial was reasonably small compared to the entire program and it agreed well with the result of Venkatesh (1995).

4.4.2. Effectiveness of the Slicing for Fault Localization

Tables 4 and 5 show the data about the time (in minutes) required to localize each of the faults in each trial. In Trial-1, with respect to G1, where each subject used the

Table 4. Data of Trial-1.

Faults / Subjects		$f1_1$	$f1_2$	$f1_3$	$f1_4$	$f1_5$	$f1_6$	$f1_7$	$f1_8$	Total
G1 (with slice)	A1	31	8	15	25	14	15	4	7	119
	A2	26	10	15	20	26	10	12	9	128
	A3	17	20	13	22	18	10	8	12	120
G2 (without slice)	B1	17	10	26	27	35	17	7	15	154
	B2	28	18	36	17	25	23	16	12	175
	B3	23	12	28	32	41	17	7	6	166

Table 5. Data of trial-2.

Faults / Subjects		$f2_1$	$f2_2$	$f2_3$	$f2_4$	$f2_5$	$f2_6$	$f2_7$	$f2_8$	$f2_9$	Total
G1 (without slice)	A1	17	6	12	30	6	11	5	5	26	118
	A2	17	5	24	13	18	16	19	5	9	126
	A3	36	24	12	41	8	15	5	4	10	155
G2 (with slice)	B1	18	6	27	18	20	20	8	2	12	131
	B2	11	8	10	16	16	13	7	7	5	93
	B3	14	14	19	36	10	5	17	1	2	118

slicing system, the average time spent to localize the faults $(f1_1) \ldots (f1_8)$ was 122 min (A1: 119 min, A2: 128 min, A3: 120 min). On the other hand, for G2, the average time was 165 min (B1: 154 min, B2: 175 min, B3: 166 min).[3] In Trial-1, the slicing function in the slicing system is therefore effective for fault localization and the subjects in G1 could detect the faults efficiently.

In Trial-2, each subject in G1 did not use the slicing tool, and the average time it took to localize the faults $(f2_1) \ldots (f2_9)$ was 133 min, whereas for G2, the average time was 114 min. The average time of G1 is greater than that of G2. However, this difference is not confirmed by the Welch test under the same level of significance. As a result, from the viewpoint of the average time taken to localize all faults, the hypothesis has not been confirmed.

4.4.3. Detailed Analysis for Individual Faults

We compared the individual fault localization time of the eight faults $(f1_1) \ldots (f1_8)$ in Trial-1 and the nine faults $(f2_1) \ldots (f2_9)$ in Trial-2. A significant difference was found between G1 and G2 for the following three faults in Trial-1: $(f1_3)$ illegal conditional statement, $(f1_5)$ lack of the procedure call, and $(f1_6)$ wrong data renovation. Also, a significant difference was found in the following two faults in Trial-2: $(f2_1)$ illegal conditional statement and $(f2_9)$ wrong registration for database.

From Tables 2, 3, 4 and 5, it can be observed that there is no correlation between the fault-localization time and the distance from the statement at which the incorrect values are printed, to that at which the fault is localized. However, for $(f1_6)$, though the distance is relatively small, there is a significant difference in fault localization time.

For $(f1_5)$, $(f1_6)$ and $(f2_1)$, the fault localization could be done easily by calculating the slice for the output variables which had wrong values. Especially, with respect to $(f1_5)$, the fault localization was easily done by comparing the two functions, since there was another function with similar processing as the faulty one.

With respect to $(f1_3)$ and $(f2_9)$, it was very difficult to localize the faults only by calculating the slice for the output variables which had wrong values. However, the subjects could localize the wrong variable among several output variables by using the trace function, because by using the slice and trace function together, the range where the fault was located can be reduced. Then, the fault could be easily detected by calculating its slice. All subjects used the trace function successfully.

As a result, the hypothesis is confirmed for the specific kinds of fault, and we suggest that slicing is effective for fault localization.

5. Second Experiment

In Experiment 1, we could not collect enough subjects to statistically confirm all hypotheses. Experiments such as Experiment 1 are very expensive in regard to control of the overall experiment process and collection of reliable data. To resolve this limitation, we have also carried out an inexpensive experiment, called Experi-

ment 2, which aimed to examine the usefulness of the slicing method to fault localization for small scale programs (of size of about 50–100 LOC) with more subjects and less management effort (the details of Experiment 2 are shown in Nishimatsu et al., 1998a).

5.1. Design of the Experiment 2

5.1.1. Goal

The goal of Experiment 2 is the same as in the first experiment.

5.1.2. Variables

In this experiment, we manipulated one independent variable.

1. The fault-localization technique used by a subject (slicing-based and conventional debugger-based).

For each fault localization, we measured one dependent variable.

1. The individual fault localization effectiveness defined as the time needed to localize the faults included in the program.

5.1.3. Design/Subjects/Process of the Experiment

Experiment 2 was conducted without using computers. There were 34 subjects, all undergraduate students of the Faculty of Engineering Science at Osaka University. They have learned Pascal programming sufficiently and had sufficient reading and experience in debugging Pascal programs. They were divided into two groups: GS1 (15 subjects) and GS2 (19 subjects).

The following six types of programs, each of which includes one fault, were prepared. The target programs are written in Pascal and are approximately 40 lines long (excluding blank lines).

(P1) Factorization,

(P2) Decision whether the input number is a prime number,

(P3) Construction of a triangle of Pascal,

(P4) Numerical operations,

(P5) Permutation, and

(P6) Sorting.

Each subject of GS1, was asked to localize the faults in the faulty programs together with slicing information which includes the same information provided from our tool by using the input as the slicing criteria. The slicing information was that the statements in the slice calculated using the most typical slicing criterion are underlined, shown in Figure 5.

Each program includes one fault. (P1), (P3) and (P6) include an illegal conditional statement, and (P2), (P4) and (P5) include an illegal assignment.

Figure 5 shows the program (P1), including the slicing information. It consists of the following information:

(I1) Function of the program.

(I2) Input, actual (wrong) output for the input and correct output for the input.

(I3) One faults: In Figure 5, the statement labeled 19 'if $k <= 0$ then', is the fault. Correct statement is 'if $k > 0$ then'.

Each subject of GS2, was asked to localize the faults only with the source program and an input/output which shows that the program includes a fault.

Table 6 shows the size of each program and the size of slice for each program.

5.1.4. Hypothesis

The hypothesis investigated in this experiment was the same as that in the previous experiment. (H1): There is a difference between subjects using slicing-based fault localization and subjects using conventional debugger-based fault localization with respect to their mean fault localization effectiveness.

5.2. Process of the Experiment

At first, the six programs are distributed to the subjects. For each program, each subject conducts the following steps:

(1) The subject understands the content of the program.

(2) The subject realizes the existence of the fault from the input and output data.

(3) The subject reads the program and localizes the fault.

The experiment is completed when fault localization for the six programs is finished.

We collected the time that each subject spent in the above steps using the original data collection form. In the form, for each program, there are three columns: start

This program outputs the result of the factorization for the input number.

Input	18
Correct output	18 = 1*2^1*3^2
Wrong output	18 = 1

Program (P1)

```
 1   program factorization(input,output);
 2   var n,f,k : integer;
 3   begin
 4      readln(n);
 5      if n<=0 then
 6             writeln('Error')
 7      else
 8      begin
 9         write(n,'=1');
10         f:=2;
11         while f<=n do
12         begin
13            k:=0;
14            while (n mod f) = 0 do
15            begin
16               n:=n div f;
17               k:=k+1
18            end;
19            if k<=0 then
20                  write('*',f,'^',k);
21            f:=f+1
22        end;
23        writeln
24      end
25   end.
```

Figure 5. Program (P1) for subjects of GS1.

Table 6. Size of program and slice in Experiment 2.

	P1	P2	P3	P4	P5	P6
LOC	25	31	51	37	49	35
Size of slice	5	7	7	13	18	18

time (the subject enters the time when he/she starts the fault localization), completed time (the subject enters the time when he/she completes the fault localization) and content of fault (the subject enters the faulty statement in the program). Thus, we can easily check the correctness of the fault localization and calculate the time for fault localization.

5.3. Result and Analysis

Table 7 shows the data of the time (in minutes) spent to localize six faults in the experiment. The average time of GS1 was 40.8 min and that of GS2 was 49.0 min. This difference was confirmed by the Welch test.

The result shows that the information about the slice is effective for the fault localization and the subjects in GS1 could detect the faults efficiently. Thus, our hypothesis has been confirmed.

6. Discussions

Here, we discuss the followings points: validity and limitation of this experiment and future research topics.

6.1. Validity of the Results

In Shimomura (1993), with respect to the types of faults, the fault-detection power of the slicing techniques has been discussed. Faults are divided into the following three types:

Table 7. Data of Experiment 2.

	GS1 (15)	GS2 (19)
P1	3.3	3.3
P2	6.5	8.1
P3	7.1	11.6
P4	5.7	4.7
P5	15.1	16.8
P6	3.1	4.5
Total	40.8	49.0

(1) Wrong-statement fault: An algorithmic or boolean expression or output argument in a statement is wrong.

(2) Wrong-name fault: The name of left-hand-side variable or the name of the input variable in a statement is wrong.

(3) Missing-statement fault: Necessary statements are missing in the program.

In Shimomura (1993), the author argues that program slicing can deal with the wrong-statement fault and the wrong-name fault. However, it cannot deal with the missing-statement fault.

In Experiment 1, among the faults which can easily detected by using slicing, ($f1_6$) and ($f2_9$) are wrong-statement faults, and ($f1_3$) and ($f2_1$) are wrong-name faults. On the other hand, ($f1_5$) is the missing-statement fault. However, in Experiment 1, the latter could be easily detected. As described in Section 4.4.3, this was mostly due to the implementation of the target programs, and it does not automatically show that conventional slicing can always deal with the missing-statement fault. Thus, we consider that it gives some evidence to the view about the fault detection power of the slicing techniques.

Here, we did not deal with the faults related to pointer types and the interaction of multiple faults. For the former matter, we understand that analysis for pointer-related faults are very important for practical program debugging. As the first step of dealing with the pointer-related faults, we have been examining the pointer-related analysis. Currently, the debug tool includes a new function to analyze the pointer and array and so it can deal with the pointer-related faults (Ashida et al., 1999). Though the analysis uses the dynamic information of program execution, we have experimentally confirmed that the cost is much smaller than the dynamic slicing calculation (Inoue et al., 2000). We would like to experimentally examine the applicability of our tool for pointer-related faults. For the latter issue, it is practically important to evaluate the usefulness of program slicing to the interaction of multiple faults. However, usefulness of program slicing to a single fault should be evaluated first in controlled experiments. As future work, we would like to evaluate the interaction of multiple faults in the practical environment.

As the qualitative evaluation, in Experiment 1, the subjects generally had good impressions for the effectiveness of program slicing to fault localization. Especially, as described in Section 4.4.3, several types of faults could be easily localized by using slicing information. On the other hand, some subjects told us that they experienced difficulty to select the appropriate slicing criterion. From the practical viewpoint, it is necessary to strengthen the ability to effectively select the slicing criterion and this is also an important future work.

In our experiments, the subjects were students belonging to computer science course of Osaka University. However, since they have studied fundamental programming techniques and have experiences of developing some programs, they have the similar programming performance of the middle standing programmer in practical environment. The programs used in the experiments are relatively small

compared to the practical software. However, the programs in Experiment 1 include fundamental stock control functions and they have been used as a standard program in computer science field in Japan. We believe the programs are suitable to the experiments. Of course, it is necessary to conduct the similar experiments under other contexts and this is an important future topic.

6.2. Target Language

As described in Section 3.1, the target language is sufficient for exploring the effectiveness of the slice; however, it would be too simple to be applied to the practical application programs. One of the important future topics is to extend the target language to the more complex-structure languages such as C, Pascal and object oriented languages.

This language has been used in an introductory course at Osaka University for 10 years. It is generally said that although several techniques proposed in the software engineering field should be taught in the academic environment, it had not been enforced sufficiently (Tichy et al., 1993). We consider that it can be used to teach the slicing-based debugging technique efficiently and thus it can be useful in the educational environment.

6.3. Application of Our System to Maintenance Processes

Based on the results of our experiments for the debugging process, it is considered that program slicing is also useful for software maintenance processes. In order to confirm this, we carried out a similar experiment which aimed to examine the usefulness of the slicing method to the maintenance process using our system (Nishimatsu et al., 1998b). In this experiment, we prepared two types of programs (PG1 and PG2) and six subjects. The subjects were divided into two groups: GM1 and GM2. The two groups were required to modify PG1 and PG2 respectively, without using the slicing technique. Then, subjects in GM1 and GM2 modify PG2 and PG1 respectively, using the slicing technique. Finally we compare the time for modification between GM1 and GM2. The results of the experiment show: (a) that program slicing technique is also effective in the software maintenance process and (b) that the slicing technique is more useful for changing the function in the original program than for adding a new function to the original program. This result also supports the usefulness of program slicing and our conclusion in this paper.

6.4. Application to Large Scale Software Development

In Experiment 1, we applied our system to a small-scale program. It was not known if the system is useful for large scale software development. However, our experi-

mental results encourage the software developers to develop similar systems and apply them to the large scale software.

In order to strengthen our system, it is necessary to develop these ideas further to solve the following problems:

(A) Selection of the target analysis part of the program and the level of abstraction.

(B) Trade-off between the precision and the calculation time of the analysis.

(C) Construction of methods to repeat the analysis and stepwise refinement.

We consider that our system is the core part of a system that could be applied to large scale software development and allow us to expect the success of the future system.

6.5. Further Development of the System

We can extend the functions of our system by adding other semantic analysis sub-tools such as other slice computation tool, symbolic execution tool, etc. We have already implemented other program slice computations (Ashida et al., 1999; Inoue et al., 2000). For example, a dynamic slicing tool has been introduced by adding a sub-tool to map the execution trace into a data dependence graph. The trace result is obtained from the interpreter shown in Figure 1. The execution trace mapped onto the dependence graph can be sent to the slicer, and the dynamic slice can be computed. The symbolic executor executes the target source program with some symbolic value inputs or some specific instance values (which causes partial evaluation of the target program).

Using these new sub-tools, we can efficiently debug the target program and improve the quality of the program and the productivity in software development.

7. Summary and Conclusions

In this paper, we have empirically evaluated the potential usefulness of program slicing to fault localization. At first, we constructed a software debugging support tool based on the static slicing method in order to conduct the experiments efficiently. Then, we conducted two kinds of experiments.

In Experiments 1 and 2, subjects (debuggers) were divided into two groups who were given programs with several faults. Subjects in Group 1 were asked to localize program faults by using the slicing-based method, whereas subjects in Group 2 localized the faults by using the conventional debugger-based method. The effectiveness of the program slice was analyzed by comparing the data collected from both groups. In Experiment 1, we aimed to perform the experiment in an environment similar to the actual software debugging. In Experiment 2, we aimed to per-

form an experiment with many subjects. These experiments confirmed that program slicing is useful for fault localization.

Based on the results of our experiments, it would be possible to improve the efficiency of the debugging process with slicing tools. For this purpose it would be essential to develop and popularize practical debugging tools with the slicing feature for overall improvement of the development process.

As the future research works, in order to evaluate the usefulness of program slicing, we are going to conduct additional experiments under the different context. In the experiments, it is necessary to take account of the pointer-related faults and multiple faults.

Acknowledgments

This research was supported in part by a grant from the Telecommunications Advancement Foundation, Grant-in-Aid for Scientific Research (C) (No:12680348) and Grant-in-Aid for Encouragement of Young Scientists (No:12780220) of MEXT.

Appendix

A. Program Semantic Analyses

In our system, the source program is represented by a graph called program dependence graph (PDG) (K.J. Ottenstein and L.M. Ottenstein, 1984), which is obtained from program dependence analysis among the statements and variables. PDG is used for the slice calculation.

A.1. Program Dependence Graph

Essentially, a PDG represents the dependences between the program statements. A node in a PDG represents a statement or a predicate of `if/while`-statement. Every arc in PDG is directed, and it represents dependence between two statements. There are two types of dependence, data dependence (DD) and control dependence (CD).

By definition, a DD from a node s_1 to s_2 means that a value defined in the statement s_1 possibly referred to the statement s_2. A CD from a node s_1 to s_2 means that the statement s_1, which is a predicate of a conditional-statement (such as if statement) or a iteration-statement (such as while statement), is dominating the execution of statement s_2.

Data Dependence

DD's can be detected by calculating reaching definition (RD) of every node. An RD of node t in a PDG is defined as a set of pair $\langle v, s \rangle$ where v is a variable and s is a statement. Pair $\langle v, s \rangle$ included in RD means that

- The value of v is defined in s, and
- Between s and t, there are no statements defining the value of v.

When RD of t includes $\langle v, s \rangle$ and v is referred to at t, we say that there is a DD from s to t.

Generally, a program code contains a number of procedure definitions. There may exist DD's through procedures using arguments and global variables. To represent these DD's in PDG, we introduce a special kind of nodes, called relay node, and extend the DD analysis to an inter-procedural one.

Control Dependence

When a conditional predicate s dominates the execution of a statement t, there exists a CD from s to t, that is, a CD represents the influence of the conditional predicate of a `if/while` statement to its internal block. CD's can be easily detected by the control flow analysis.

Building PDG

A PDG is built in several steps: program syntax analysis, mapping of statements and PDG nodes, calculation of RD for each statement, calculation of CD and DD. For the detail of every step (Ueda et al., 1993; Takaya, 1994).

We show an example of a program and its PDG in Figures 6 and 7, respectively. In Figure 7, a rectangle represents a statement in the program, and an oval represents a relay node to a procedure. A labeled arc represents a DD, and a dotted arc represents a CD. The label of a DD arc denotes the name of a variable whose value is propagated by this relation. Each procedure definition (represented as a sub-graph of the PDG) is enclosed by dotted lines.

A.2. Program Slice

A program slice is calculated by traversing a PDG. There are two types of slicing – forward slicing and backward slicing. For both cases, the starting node (called slicing criterion) is designated in the PDG at first.

A forward slice is calculated by gathering the successors of a starting node, which can be reached by traversing both DD's and CD's in their forward directions. On the other hand, a backward slice is calculated by gathering the predecessors of the starting node, which can be reached by traversing the arcs in their reverse directions.

In Figure 6, underlined statements are the result of the backward slice calculated from the variable g in line 36. For instance, statements for function 1 cm are not included in the slice result, since they are not necessary to define the value of g.

```
 1  program euclid(input,output);
 2    var x,y,g,l:integer;
 3    function gcd(m,n:integer):integer;forward;
 4    procedure swap(var a,b:integer);
 5      var     temp:integer;
 6      begin
 7        temp:=a;
 8        a:=b;
 9        b:=temp;
10      end;
11    function lcm(a,b:integer):integer;
12      var     c:integer;
13      begin
14        c:=gcd(a,b);
15        lcm:=(a div c)*(b div c)*c
16      end;
17    function gcd;
18      var     w:integer;
19      begin
20        if m < n then begin
21          swap(m,n);
22        end;
23        while n<>0 do begin
24          w:=m mod n;
25          m:=n;
26          n:=w;
27        end;
28        gcd:=m;
29      end;
30    begin
31      writeln('Input x and y');
32      readln(x,y);
33      writeln('x=',x,' y=',y);
34      g:=gcd(x,y);
35      l:=lcm(x,y);
36      writeln('gcd=',g);
37      writeln('lcm=',l);
38    end.
```

Figure 6. Example program source code.

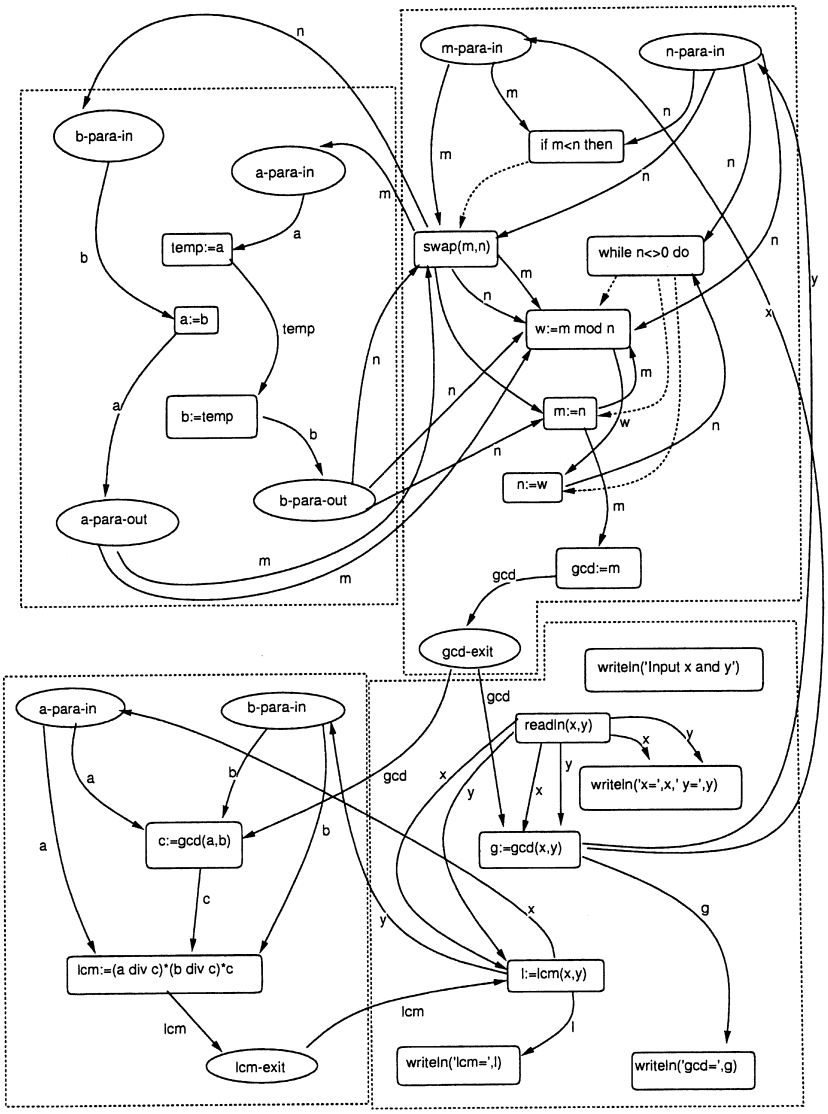

Figure 7. Example program dependence graph.

Notes

1. A failure means the inability of a system or component to perform its required functions with specified performance requirements (IEEE Standard Glossary of Software Engineering Terminology)
2. A fault means an incorrect step, process or data definition in a computer program (IEEE Standard Glossary of Software Engineering Terminology). The terms 'error' and 'bug' are also used to express fault in this paper.

3. This difference was confirmed by the Welch test with a level of significance of 0.05 (Shiba et al., 1984). Hereafter, for all the tests, the level of significance was 0.05.

References

Agrawal, H., and Horgan J. R, 1990. Dynamic Program Slicing, Proceedings of *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*. New York: ACM Press. 246–256.

Agrawal H., Demillo, R. A., and Spafford, E. H. 1993. Debugging with Dynamic Slicing and Backtracking. *Software: Practice and Experience* 23(6): 589–616.

Ashida, Y., Ohata, F., and Inoue, K. 1999. Slicing Methods Using Static and Dynamic Information. *Proceedings of the sixth Asia Pacific Software Engineering Conference* 344–350.

Basili, V. R., 1997. Evolving and Packaging Reading Technologies. *Journal of Systems and Software* 38(1): 3–12.

Basili, V. R., Caldiera, G., and Rombach, H. D. 1994. Goal Question Metric Paradigm. In: John J. Marciniak (ed): *Encyclopedia of Software Engineering*. vol. 1, John Wiley & Sons. 528–532.

Collofello J. S., and Woodfield, S. N. 1989. Evaluating the Effectiveness of Reliability-Assurance Techniques. *Journal of Systems and Software* 9(3): 191–195.

Frakes, W. B., Fox, C. J., and Nejmeh, B. A. 1991. *Software Engineering in the UNIX/C Environment*. Englewood Cliffs, New Jersey, USA: Prentice-Hall.

Fritzson, P., Gyimothy, T., Kamkar, M., and Shahmehri, N. 1991. Generalized Algorithmic Debugging and Testing. *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*. 317–326.

Humphrey, W. S. 1989. *Managing the Software Process*. Software Engineering Institute, Addison-Wesley, Reading, MA.

IEEE Standard Glossary of Software Engineering Terminology. 1990. IEEE, ANSI/IEEE Std 610.12-1990.

Inoue, K., Ohata F., and Ashida, Y. 2000. Lightweight Semi-dynamic Methods for Efficient and Effective Program Slicing. Technical Report of Osaka University, Department of Information and Computer Sciences, IIP Lab, IIP-06-05-00.

Korel, B., 1988. PELAS-Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, 14(9): 1253–1260.

Korel, B., and Laski, J. 1990. Dynamic Slicing of Computer Programs. *Journal of Systems and Software* 13: 187–195.

Lian, L., Kusumoto, S., Kikuno, T., Matsumoto, K., and Torii, K. 1997. A New Fault Localizing Method for Program Debugging Process. *Information and Software Technology* 39: 271–284.

Lyle, J. R., and Weiser, M. 1987. Automatic Program Bug Location by Program Slicing. *Proceedings of second International Conference on Computers and Applications*, 877–882.

Myers, G. J., 1979. *The Art of Software Testing*. Wiley-Interscience.

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

Nishimatsu, A., Kusumoto, S., and Inoue, K. 1988b An Experimental Evaluation of Program Slicing on Software Maintenance Process. Technical Report of IEICE SS97-87, pp. 79–86 (in Japanese).

Ottenstein, K. J., and Ottenstein, L. M. 1984. The Program Dependence Graph in a Software Development Environment. In: *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Notices* 19(5): 177–184.

Sato, S., Iida, H., and Inoue, K. 1996. Software Debug Supporting Tool Based on Program Dependence Analysis. *Transactions. on IPSJ* 37(4): 536–545 (in Japanese).

Shiba, Y., Watanabe, H., and Ishizuka, T. 1984. *Statistical Dictionary*. Shinyousha (in Japanese).

Shimomura, T. 1993. Critical Slice-Based Fault Localization for Any Type of Error. *IEICE Transactions on Information and Systems* E76-D (6): 656–667.

Takaya, N. 1994. Program Simplification Method Based on I/O Restriction. Master dissertation, Osaka University.
Tichy, W. F., Harbermann, N., and Prechelt, 1993. Future Directions in Software Engineering. *ACM SIGSOFT, Software Engineering Notes* 18(1): 35–48.
Ueda, R., Lian, L., Inoue, K., and Torii, K. 1993. Dependence Analysis and Program Slicing of Recursive Programs. Technical Report of IEICE, SS93-24, pp. 33–40.
Venkatesh, G. A. 1995. Experimental Results from Dynamic Slicing of C Programs. *ACM Transactions on Programming Languages and Systems* 17(2): 197–216.
Weiser, M. 1982. Programmers Use Slices When Debugging. *Communications of the ACM* 25(7): 446–452.
Weiser M. 1984. Program Slicing. *IEEE Transactions on Software Engineering* 10(4): 352–357.

Shinji Kusumoto received the B.E., M.E. and D.E. degrees in information and computer sciences from Osaka University in 1988, 1990 and 1993, respectively. He is currently an Associate Professor at Osaka University. His research interests include software metrics, software quality assurance technique. He is a member of the IEEE, IPSJ, IEICE and JFPUG.

Keisuke Nishie received the B.E. and M.E. degrees in information and computer sciences from Osaka University, Japan, in 1997 and 1999, respectively. He currently belongs to Matsushita Communication Industrial Co., Ltd.

Akira Nishimatsu received the B.E. and M.E. degrees in information and computer sciences from Osaka University, Japan, in 1997 and 1999, respectively. He currently belongs to R3 Institute.

Katsuro Inoue received the B.E., M.E. and Ph.D. in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was assistant Prof. of University of Hawaii at Manoa in 1984-1986. He was research associate of Osaka University in 1984-1989, and assistant professor in 1989-1995, and professor from 1995. His interests are various topics on software engineering such as software process modeling, program analysis, and software development environment. He is a member of IEEE, ACM, IPSJ, IEICE and JSSST.