Martha Hansard, Roger Lamprey, Arthur Nevins, Mark Rice, William Richardson, and James Senn for invaluable help at various stages of the work reported in the paper.

REFERENCES

1. G. D. Bergland, "A Guided Tour of Program Design Methodologies," Computer, Vol 12 (1981), 13-37.
2. R. E. Brooks, "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology," Communications of the ACM, Vol 23 (1980), 207-213.
3. Bill Curtis, IEEE Tutorial: Human Factors in Software Development, IEEE, New York, 1981.
4. Bill Curtis, "Substantiating Programmer Variability," Proceedings of the IEEE, Vol 69 (1981), 846.
5. T. R. G. Green, "Conditional program statements and their comprehensibility to professional programmers," Journal of Occupational Psychology, Vol. 50 (1977), 93-109.
6. T. D. Korson, "An Empirical Study of the Effects of Modularity on Program Modifiability," Ph.D. dissertation proposal, Georgia State Univ., unpublished, 1985.
7. B. Lientz, "Issues in Software Maintenance," ACM Computing Surveys, Vol 15 (1983), 271-278.
8. L. T. Love, "Relating individual differences in computer programming performance to human information processing abilities," Ph.D. Dissertation, University of Washington, Unpublished, 1977.
9. H. C. Lucas and R.B. Kaplan, "A Structured programming experiment," The Computer Journal, Vol. 19 (1974), 136-138.
10. D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," Communications of the ACM, Vol 15 (1972), 1053-1058.
11. D. L. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, Vol 5 (1979).
12. Svend Ryge, "Evaluating Structured COBOL as a Software Engineering Discipline," Data Base, (1981), 3-6.
13. H. Sachman, W.T. Ericson, and E.E. Grant, "Exploratory experimental studies comparing online and offline programming performance," Communications of the ACM, Vol 11 (1968), 3-11.
14. B. A. Sheil, "The Psychological Study of Programming," Computing Surveys, Vol 13 (1981), 101-120.
15. S. Shepard, B. Curtis, P. Milliman, and T. Lowe, "Modern Coding Practices and Programmer Performance," Computer, Vol. 12 (1979), 41-49.
16. Ben Shneiderman, R. Miara, J. Musselman, and J. Navarro, "Program Indentation and Comprehensibility," Communications of the ACM, Vol. 26 (1983), 861-867.
17. M. E. Sime, T.R.G. Green, and D.J. Guest, "Psycological evaluation of two conditional constructions used in computer languages," International Journal Man-Machine Studies, Vol. 5 (1973), 123-143.
18. E. Sime, T.R.G. Green, and D.J. Guest, "Scope marking in Computer Conditionals - A Psychological evaluation," International Journal Man-Machine Studies, Vol. 9 (1977), 107-118.
19. E. Soloway, J. Bonar, and Kate Ehrlich, "Cognitive Strategies and Looping Constructs: An Empirical Study," Communications of the ACM, Vol. 26 (1983), 853-860.
20. Stevens, Myers, and Constantine, "Structured Design," IBM Systems Journal, Vol 13 (1974), 115-139.
21. Iris Vessey and Ron Weber, "Some Factors Affecting Program Repair Maintenance: An Empirical Study," Communications of the ACM, Vol 26, (1983), 128-134.
22. Iris Vessey and Ron Weber, "Research on Structured Programming: An Empiricist's Evaluation," IEEE Transactions on Software Engineering, Vol SE-10 (1984), 397-407.
23. L. Weissman, "Psychological Complexity of Computer Programs: An Experimental Methodology," SIGPLAN Notices, Vol. 9 (1974), 25-36.

CHAPTER 13

Experiments on Slicing-Based Debugging Aids*

Mark Weiser
Jim Lyle
Computer Science Department
University of Maryland
College Park, MD 20742

ABSTRACT

Programming slicing is a method for reducing the amount of code looked at when debugging or understanding programs. Previous work concentrated on showing that programmers mentally slice during debugging. We present new work which concentrates on evaluating automatic tools for presenting slices to the debugging programmer. For one such tool, an online window-based editor/compiler/slicing system, we were unable to show that slicing helped. A second experiment, pencil and paper this time, presented programmers with *dices* of programs. A dice is a slice on incorrect variables from which slices on correct variables have been removed. Programmers using the dicing tool debugged their programs significantly faster than unaided programmers.

1. Introduction

Debugging and maintaining computer programs, especially programs written by someone else, is a difficult and time consuming task. Among other things, the programmer needs to understand how a program produced a particular result so that the program behavior can be modified.

One aid to understanding is to reduce the amount of detail a programmer sees by extracting only relevant information. An application of data-flow analysis, program slicing, can be used to transform a large program into a smaller one containing only those

*Research supported in part by AFOSR contract #82-0303 and by the NASA Goddard Human Factors Committee.

statements relevant to the computation of a given output [12].

A previous experiment showed that programmers mentally construct slices when debugging[11]. However, in spite of some preliminary experiments[9, 5] the efficacy of slicing-based tools remains to be proven. We present here studies of debugging with the aid of two different automatic slicing tools.

2. Debugging Tools

Psychological theories of debugging and tools for debugging are not often linked. Before describing the tools we built based on slicing theory it is useful to review the range of other debugging tools, including that most common one: no tool at all.

2.1. Debugging Without Tools

A study of software engineering practices of 30 companies found only 27 percent using some type of testing tool[14]. They attributed the low usage to several factors, among them a perceived high cost of tool usage, managers lacking software experience, and previous experience with incomplete and poorly documented tools. One is therefore not surprised to find that often the reaction to the question, "are there any debugging tools available on our computer" is not "yes, and it works great," but instead "well, yes, but you don't want to use it." This somewhat negative reaction to debugging tools and experimental results [3] that suggest code reading may be the most effective debugging approach, has lead some authors, Myers, for example, to advocate avoiding automatic tools except as a last resort [8]. What Myers does advocate is a cycle of think, generate hypotheses from the program behavior and other known facts, and test the generated hypotheses until the error is found.

2.2. Debugging With Tools

Debugging tools have evolved through three generations from the first memory dump programs to sophisticated systems that sometimes can locate faulty statements.

The first generation of tools provides information in terms of the underlying machine architecture. These tools, often called *low level debuggers*, are the memory dumps and absolute instruction traces found on most systems. An example of an interactive first generation tool is the Unix adb[7].

The second generation of tools provides information in terms of the programming language used to write the program. The information provided by these tools, often called *high level debuggers* or *symbolic debuggers*, is available immediately for the programmer's use. He does not need to determine answers to questions like "what is the memory address of variable X", so that he can examine the value of X from the memory dump. Examples of this kind of tool are the Unix sdb and dbx, VAX DEBUG[1], and the PLUM system[13].

The third generation of tools does more than provide raw information. These tools try to make some deduction about the presence and location of faults in the program. Two examples of third generation tools are DAVE [2] a tool that finds likely errors by examining the data-flow relationships of variables, and FALOSY [10] a debugging expert-system.

A fourth kind of system, not so much a debugging tool as a tool for understanding bugs, is represented by the Proust system[4]. Systems like Proust could eventually evolve into intelligent debugging aids.

3. Debugging With Program Slices

A program slice is computed on a set of variables at a given statement. The resulting subset of program statements are all statements relevant to the computation of the set of variables at the given statement.

Since a program slice contains all statements that could have influenced the value of a variable at some statement, if the printed value of some variable is incorrect then the *bug* should be evident somewhere in the slice on that variable at that print statement (but see exception, below). As an example, consider the following program intended to compute some simple statistics.

The right hand side of line 8 of the program in Figure 1 should have "sum + x(i)" instead of "x(i)." If we have a *reliable* testing method, i.e., one that shows the existence of faults, then we would discover through testing that the value computed for *avg* was incorrect. At this point the usual approach is to examine the entire program to try to locate the fault. Using program slicing we would first compute the slice on *avg* at line 16 and examine the resulting program slice for the fault. Figure 2 presents the slice on *avg* at line 16. All reference to the *std* computation that is irrelevant to the computation of *avg* has been removed so that the programmer can examine a smaller program in search of the fault.

A slice computed at the output statement on an incorrectly valued output variable does not always contain the fault. For example, if the statement at line 8 in the program of Figure 1 were changed to:

```
8  20   ssq = sum + x(i)
```

then a slice (see Figure 3) on *avg* at line 16 would not contain line 8, the incorrect statement. However, comparison of the slice on *avg* with the program specifications in Figure 3 would show that the computation of *avg* does not meet the specifications. Further slicing would cast suspicion on lines 7 and 8 since these lines do not appear in the slice of any output variable. Also, data-flow analysis would show that the value of *ssq* computed by lines 7 and 8 is never used (line 9 assigns *ssq* before the value could be used) and hence an error should be suspected on those lines.

4. A slicing environment – no significant advantage

Our first experiment evaluating slicing-based aids had no statistically significant results. It is nonetheless a useful lesson in the problems of evaluating tools, and so worth discussing briefly. More details are in Lyle's thesis[6].

In evaluating a new tool the Hawthorne effect is a constant menace. In the short time of the experiment subjects may react more to the newness of the tool than to any specific quality. To control for this we created a completely new multi-window programming environment called Focus. Subjects performing both slicing and non-slicing tasks were required to work in Focus, which had its own editor, compiler, and user interface (including help system). The only user interface difference between treatments is that during some tasks users had an extra command, *slice*, in their pop-up menu.

In spite of this careful control, a randomized order within-subjects design, and learning trials, we were unable to show that having a slicing tool helped reduce debugging time. If anything the trend was the other way: slicer users took a little longer to debug, possibly because they were playing with their new command.

This experimental result was disappointing because it had seemed to follow from the mental use of slices that a slicing aid would be useful. Perhaps more learning time, or larger programs (ours were around 100 lines) would have given a different result. In post-experiment interviews subjects did say they liked having the slicing command. Or, perhaps, slicing is like watching a beautiful sunset – a computer *can* do it, but it just isn't the same.

Figure 1

Stat: Program to Compute Average and Standard Deviation

```
1           subroutine stat(n,avg,std)
2           real x(20)
3           read (8,100)n
4           do 10 i = 1,n
5     10    read (8,200)x (i)
6           sum = 0
7           do 20 i = 1,n
8     20    sum = x (i)
9           ssq = 0
10          do 30 i = 1,n
11    30    ssq = ssq + x(i)**2
12          avg = sum/n
13          std = sqrt(ssq - n*avg**2)/(n-l))
14          print 300,n
15          print 600,(x(i),i = 1,n)
16          print 400,avg
17          print 500,std
18          return
19    100   format (15)
20    200   format (f10.0)
21    300   format ('n =',15)
22    400   format ('avg =',f10.4)
23    500   format ('std =',f10.4)
24    600   format ('x = ',15f4.0)
25          end
```

Figure 2

Slice from Stat on avg at Line 16

```
1           subroutine stat(n,avg,std)
2           real x(20)
3           read (8,100)n
4           do 10 i = 1,n
5     10    read (8,200)x(i)
6           su, = 0
7           do 20 i = 1,n
8     20    sum = x(i)
12          avg = sum/n
16          print 400,avg
18          return
19    100   format (i5)
20    200   format (f10.0)
22    400   format ('avg = ',f10.4)
25          end
```


Figure 3

Slice from Incorrect Program Does Not Contain Fault

(Slice on avg at 16)

```
1         subroutine stat(n,avg,std)
2         real x(20)
3         read (8,100)n
6         sum = 0
12        avg = sum/n
16        print 400,avg
19  100         format (i5)
22  400 format ('avg =',f10.4)
25        end
```

Figure 4

Specifications for Program of Figure 3

INPUT:

n Number of elements in array X

X Array of n real numbers

OUTPUT:

n Number of elements in array X

avg Sum of elements in X divided by n

std Standard deviation of elements in X

5. Evaluation of Dicing – it seems to help

Our second experiment was more successful. Previous work on using slicing fo debugging uses only the information that some variables are computed incorrectly. I does not use the other information gained from testing that some variables might b correct.

5.1. Introduction to Dicing

We propose a method, called *dicing, which combines multiple slices to further refin the location of* program faults. Dicing is a heuristic only, but in practice [6] seems to be a good one. Dicing starts like slicing, with a slice on variables with incorrect values. The fault is likely in this slice. Dicing then separately slices on variables with correct values, and makes the assumption that the fault is unlikely to be in this slice. Combining these two slices gives the basic dice: the slice on incorrect variables *less* the slice on correct variables.

In discussing dicing it is convenient to distinguish two sets of variables. The first set, called *KBI* (known to be incorrect) contains all the variables which testing has identified as containing incorrect values. The second set, call *CSF* (correct so far) contains variables which testing has identified as correct. *Dicing* is the following procedure: (1) slice on KBI. (2) slice on CSF. (3) Remove from slice (1) the statements in slice (2). The result is a dice. (More details are in Lyle [6]).

Dicing depends on three assumptions:

(1) Testing has been reliable and all incorrectly computed variables have been identified.

(2) If the computation of a variable, *v*, depends on the computation of another variable, *w*, then whenever *w* has an incorrect value then *v* does also.

(3) There is exactly one fault in the program.

The three assumptions are necessary for the correct operation of dicing. If testing has not identified all incorrectly computed variables then we could have the following situation: A single fault could cause two variables to be incorrectly valued while unreliable testing identifies only one variable as a member of KBI and incorrectly places the other variable in CSF. If these two variables are used for dicing the faulty statement would not be included in the dice.

If the second dicing assumption does not hold then variables that depend on the faulty statement could be placed in CSF and hence dicing would fail to include the faulty statement. This could also happen if a second fault canceled the incorrect value for some of the output variables.

As an example of dicing, consider the program fragment in Figure 5.

If the intended output for Y is $2A^2$ and the intended value for Z is A^2-2 instead of A^2+2, we would go through the following steps to isolate the fault to line four.

(1) Execute the program with reliable test data. The result of such a test would be that the value for Z was not correct and that the value for Y was correct.

Figure 5
Dicing Example

```
1    Get (A);
2    L := A**2;
3    Y := L*2;
4    Z := L + 2;
5    Put (Y,Z);
```

(2) Slice on the incorrectly valued variable Z, at line 5. This restricts our attention to lines, 1,2,4, and 5.

(3) Slice on the correctly valued variable, Y at line 5. This identifies lines 1,2,3, and 5 as being correct.

(4) Considering the specification and the two slices together, the dice is statement 4, which contains minus instead of plus in the expression.

We are left with three issues:

(1) In general, how should we select the variables for slicing and dicing?

(2) How useful is dicing when the above assumptions are relaxed?

(3) How useful is dicing when people try to use it?

(1) and (2) are addressed in [6]. An experiment to evaluate (3) is discussed below.

5.2. An Experiment With Dicing

We evaluated how well programmers can use the information gained from dicing by introducing three faults into a program and timing how long ten programmers took to find the error, given the listing of a slice on a KBI variable with the dice highlighted. These times were compared to the times of ten programmers given the entire program listing without slicing as a debugging aid.

5.2.1. Hypotheses

Our central hypotheses is that programmers using the dicing information find faults faster than programmers using traditional methods. We also would like to check the hypothesis that there is no significant difference in the debugging ability of the control and experimental groups. Debugging ability is difficult to accurately measure so we tested the hypothesis that the background and experience between the two groups was not significantly different.

5.2.2. Subject Selection

We recruited subjects from the computer science graduate student population at the University of Maryland. We also recruited some subjects from the University of Maryland Computer Science Center's System Support staff and one physics graduate student with many years of FORTRAN experience.

Each subject was asked the following background questions:

(1) How long have you been programming?

(2) How many CMSC classes in your BS/BA?

(3) How many other CMSC classes?

(4) What programming languages are you familiar with?

(5) On a scale from 0 to 10, how good are you with FORTRAN?

We found that the subjects had a wide range and variety of experience. Table 1 summarizes the subject's responses.

Table 1
Background Summary

name	N	mean	sd	min	max	median
Experimental Group Background						
years	10	9.7	6.4	2.0	19.0	7.5
bs-classes	10	7.2	4.4	0.0	13.0	7.5
other-classes	10	7.6	6.4	0.0	20.0	9.5
tot-classes	10	14.9	7.5	3.0	25.0	15.5
languages	10	5.9	2.5	3.0	12.0	5.5
skill	10	6.7	2.4	2.0	10.0	7.5
Control Group Background						
years	10	8.3	3.4	3.0	13.0	9.0
bs-classes	10	7.4	5.7	2.0	20.0	6.0
other-classes	10	4.3	5.8	0.0	15.0	1.0
tot-classes	10	11.7	8.7	2.0	27.0	8.5
languages	10	8.5	4.1	1.0	15.0	9.0
skill	10	7.2	2.6	2.0	10.0	7.0
Total Group Background						
years	20	9.0	5.0	2.0	19.0	9.0
bs-classes	20	7.3	5.0	0.0	20.0	6.5
other-classes	20	5.9	6.2	0.0	20.0	4.5
tot-classes	20	13.3	8.1	2.0	27.0	13.0
languages	20	7.2	3.6	1.0	15.0	6.5
skill	20	6.9	2.5	2.0	10.0	7.0

All subjects had at least two years experience programming and most subjects had at least nine. All subjects claimed to "know some FORTRAN", and most subjects claimed to feel "comfortable with FORTRAN," the language of the programs in the experiment. Table 2 shows how many subjects claimed

to be familiar with the given language. Some languages such as ALGOL and Forth had only one or two claimants and were omitted. In Table 2, assembler means any assembly language. Fifteen subjects were familiar with at least one assembler and six subjects with two or more. One likely difference between these subjects and real world programmers is that if the distribution of language familiarity for the subjects were compared with real world programmers, the frequency of PL/I and COBOL would be higher and the distribution of Lisp and C would be less.

A Mann-Whitney statistic, $U < 23$, is significant at the 0.05 level for a two tailed test with ten subjects in each group. As Table 3 shows, we found no significant difference between the background of the experimental and the control group subjects. We feel that we can conclude that any difference in performance between the two groups would be from the treatments applied to the groups.

Table 2
Language Frequency

Language	Number of Subjects
FORTRAN	20
Pascal	16
Assembler	15
C	13
Lisp	10
Basic	8
COBOL	6
SNOBOL	6
APL	6
PL/I	5

Table 3
Mann-Whitney U for Background

Variable	U	Low Group
years	48.0	cont
bs-classes	45.5	cont
other-classes	37.5	cont
tot-classes	39.5	cont
languages	28.0	exp
skill	43.0	exp

5.2.3. Procedures

The subject first answers a background questionaire and takes a practice treatment. They then are given an explanation of the program to be debugged and finally locate each of three bugs in a FORTRAN program. The goal of the practice treatment is to get the subject in the frame of mind for debugging, and remove any confusion about what he should be doing.

The practice treatment for the experimental group is an explanation of slicing and dicing and why they should help locate a program fault. The subject then locates a fault in a FORTRAN program that computes mean and standard deviation.

The control group is told that the experimenter is collecting data on how programmers debug programs and would he debug a small program, explaining what he is doing and thinking as he goes.

After the practice treatment, the experimenter gives an informal specification of the program to be debugged to both groups of subjects. The subject and experimenter discuss a sample correct output, variable naming conventions and the general algorithm used in the program until the subject understands the materials.

The procedure for measuring fault location time is to identify to the subject a variable as having been shown incorrect by testing. The subject is given (face down) a listing of the relevant slice with the diced set highlighted (the control group received a listing of the entire program). Timing begins when the subject turns over the listing and ends when he states that he has found the fault. If the subject has not correctly identified the fault, he is so informed and timing continues until the fault is correctly identified.

5.2.4. Materials

The experimental materials used are a survey form, the listing of a practice program, a description of how to use slicing and dicing for debugging, documentation to a FORTRAN program to compute statistics on letters, words, lines and sentences in a file, a listing of this program with three planted faults, and the listing of a program slice for each fault.

The practice program is a mean and standard deviation program written in FORTRAN with a fault installed in the initialization of the variable *sum*. This program was chosen because it is a practical program, easy for the subjects to understand, and yields clear slices. This program also refreshes the subject on statistics used in the experimental program so that, some learning (or maybe relearning) effects are removed.

Only the experimental group receives a description of how to use slicing and dicing along with sample slices from the practice program.

The documentation includes a description of what the program is supposed to do, a sample input, a sample correct output for the given input, and a table of major variables and their meaning. Since all three faults introduced into the experimental program involve minimums or maximums, included in the description is a generic explanation of how to compute a minimum or maximum. We want the subject to know what he is looking for; we do not want to measure how long it takes for him to remember how to compute a minimum.

The experimental program reads a file of text and computes descriptive statistics on the letters, words, lines, and sentences in the file. Since many algorithms follow a pattern of initialization, intermediate computation, result, we installed in the experimental program a fault in each one of these locations. A fault was placed in the result phase of the computation of the variable XWTS (maximum words per sentence), the intermediate computation phase of MLETTL (minimum letters per line), and the initialization of MWTL (minimum words per line).

5.3. Experimental Results

Warm-up is the time spent from beginning to fill out the questionaire until the subject is ready to start looking for the faults in the experimental program. Total-time is the total time spent looking for faults (XWTS + MLETTL + MWTL). The Table 4 summarizes the raw data on debugging times.

The Table 5 presents Mann-Whitney U statistics for testing the null hypothesis that there is no difference in performance between the two groups. A U value < 23 is significant at the 0.025 level for a one-tailed test. As shown in Table 5, we found a significant difference between the experimental and control group performance on two of the three faults and for the time to locate all the faults.

Table 4
Debug Times Summary

Name						
Experimental Group						
XWTS	10	97.4	131.3	22.0	461.0	53.5
MLETTL	10	162.8	106.2	69.0	413.0	123.0
MWTL	10	19.0	16.2	6.0	54.0	12.0
Warm Up	10	761.1	237.2	453.0	1160.0	744.0
Total Time	10	279.2	239.4	111.0	928.0	207.0
Control Group						
XWTS	10	192.9	112.3	54.0	364.0	174.0
MLETTL	10	309.9	181.1	87.0	651.0	267.0
MWTL	10	67.2	97.7	7.0	312.0	23.5
Warm Up	10	642.7	124.0	416.0	879.0	630.5
Total Time	10	570.0	257.3	273.0	1026.0	508.5
Total Group						
XWTS	20	145.1	128.6	22.0	461.0	85.0
MLETTL	20	236.4	163.0	69.0	651.0	192.5
MWTL	20	43.1	72.5	6.0	312.0	17.0
Warm-Up	20	701.9	194.2	416.0	1160.0	641.5
Total-Time	20	424.6	284.2	111.0	1026.0	325.0

It should be pointed out that the non-statistically significant fault (MWTL) is on the boundary of the critical region (U < 27) for the 0.05 level of significance, and that the median time to locate this fault was less than 20 seconds. We observed that the subjects who took the longest to find this fault were members of the control group using a backward search method. Since the fault was with the initialization of MWTL, the subjects took much time working back to the beginning of the program.

Table 5
Mann-Whitney U for Debugging Times

Variable	U	Low Group
XWTS	19.0	exp
MLETTL	20.0	exp
MWTL	27.5	exp
Total-Time	13.0	exp
Warm-Up	37.0	control

6. Summary and Conclusions

A pencil and paper based simulation of a slicing-based tool was shown to be useful. Despite great effort at designing a homogeneous interactive environment and data collection tool, a slicing tool in that environment could not be shown useful. Possibly the novelty of that environment added enough noise to make any differences among individual tools. Others attempting to measure tools in new environments should be wary of the environmental noise. Dicing was quite successful and may one day be found in every programmer's toolkit.

REFERENCES

[1] Bert Beander. "VAX DEBUG: An interactive, symbolic multilingual debugger." *SIGPLAN Notices* **18**, 8, pp. 173-179, August 1983. (Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging)

[2] L. D. Fosdick and L. J. Osterweil. "Data flow analysis in software reliability." *ACM Computing Surveys* **8**, 3, pp. 305-330, September 1976.

[3] J. D. Gould and P. Drongowski. "An exploratory study of computer program debugging." *Human Factors* **1**, 6, pp. 258-277, 1974.

[4] W. L. Johnson and E. Soloway. "PROUST: Knowledge-Based Program Understanding." *IEEE Transactions on Software Engineering* **SE-11**, 3, pp. 267-275, March 1985.

[5] Herbert D. Longworth. Sliced-based Program Metrics. M.S. Thesis, Michigan Technological University 1985.

[6] J. Lyle. "Evaluating Variations on Program Slicing for Debugging." Ph.D Dissertation, Computer Science Dept University of Maryland, Dec 1984.

[7] J. F. Maranzano and S. R. Bourne. "A tutorial introduction to Adb." pp. 323-350 in *Unix Programmer's Manual Volume 2*, Holt, Rinehart, and Winston, 1983.

[8] Glenford J. Myers. *The Art of Software Testing.* John Wiley and Sons, 1979.

[9] K.J. Ottenstein and L.M. Ottenstein. "The Program Dependence Graph in a Software Development Environment." pp. 177-184 in *Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium of Practical Software Development Environments*, ed. P. Henderson, ACM SIGPLAN, SIGSOFT Engineering Notes, Pittsburgh, Penn., April 23-25, 1984.

[10] Robert L. Sedlmeyer and William B. Thompson. "Knowledge-based fault localization in debugging." *SIGPLAN Notices* **18**, 8, pp. 30-41, August 1983. (Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging)

[11] Mark Weiser. "Programmers Use Slices When Debugging." *Communications of the ACM* **25**, 7, pp. 446-452, July, 1982.

[12] Mark Weiser. "Program slicing." *IEEE Transactions on Software Engineering* **SE-10**, 4, pp. 352-357, July 1984.

[13] Marvin V. Zelkowitz. *PL/I Programming with PLUM.* Paladin House, Geneva, Illinois, 1976.

[14] Marvin V. Zelkowitz, Raymond T. Yeh, Richard G. Hamlet, John D. Gannon, and Victor R. Basili. "Software engineering practices in the US and Japan." *Computer* **17**, 6, pp. 57-65, June 1984.