The Relationship of Slicing and Debugging to Program Understanding

Margaret Ann Francel
Dept. of Mathematics and Computer Science
The Citadel
francelm@citadel.edu

Spencer Rugaber
College of Computing
Georgia Institute of Technology
spencer@cc.gatech.edu

Abstract

The paper describes a study that explores the relationship between program slicing and code understanding gained while debugging. The study consisted of an experiment that compared the program understanding abilities of two classes of debuggers: those who slice while debugging and those who do not. For debugging purposes, a slice can be thought of as a minimal subprogram of the original code that contains the program faults. Those who only examine statements within a slice for correctness are considered slicers; all others are considered non-slicers. Using accuracy of reconstruction as a measure of understanding; it was determined that slicers have a better understanding of the code after debugging.

Keywords: Program slicing, reverse engineering, debugging

1. Introduction

Debugging code comprises a significant portion of the software development and maintenance process. Yet no “best” method for debugging programs is known. However, the experiment described in this paper indicates that debuggers who use slicing while debugging have a better understanding of the code at the end of the debugging process than those who do not use slicing.

Debugging is the task of identifying faults in code. A goal of the *debugger*, the person who debugs the code, is to localize the fault area of the code and, at the same time, to develop an understanding of the program so that an adequate correction can be made. Working toward this goal is a labor-intense and time-consuming activity. As a result it is critical to identify debugging strategies that lead to quick reduction of the code fault area coupled with increased understanding.

Above, and throughout the paper, we use the term fault as defined in the *IEEE Standard Glossary of Software Engineering Terminology* [2]. The glossary distinguishes between a program error, fault and failure. A mistake in the human thought process made during the construction of a program is called an *error.* Evidence of errors comes through program *failures*, typically incorrect output values, unexpected program termination, or nonterminating execution. It is often the case that the root cause of a failure can be traced to a small area of a program. If so, that area is said to contain a *fault*. It is important to note that sometimes program failures are indications of global problems such as mistaken assumptions or inappropriate architectural decisions. In such cases, it is misleading to assume that editing a small area of the program will prove sufficient to correct an error.

At the start of debugging, the code fault area can be anywhere in a program. It is the task of the person who debugs the code to reduce this range as much as possible. Unfortunately, no one method of code reduction is favored universally by people who debug programs. Rather different people prefer different methods. However, a large number of experts use a code reduction method called program slicing while debugging [8].

The concept of slicing was developed by Mark Weiser in the early 1980’s [7]. Slicing is based on a program’s flow of data. Formally a *slice* of code with respect to a statement S_i and variable x_j consists of exactly those statements of the code that might affect the value of x_j at statement S_i. Figure 2 shows three slices taken from the buggy C++ program shown in Figure 1.

For emphasis in Figure 2 below statements that are not contained in all three slices are bolded.

If the variable x_j in the output statement S_i contains an incorrect value, then the slice on x_j at S_i will contain the program fault causing the incorrect output value. This is true because a slice on variable x_j with respect to statement S_i contains all statements that might affect the value of x_j at S_i, and program failures are indications of program faults. This makes slices based on statements that are outputting incorrect values especially helpful in debugging. For example, in the buggy program of Figure 1, once one determines that exactly one output value is incorrect and that that value is being output in statement #41, the code fault area of the program can be reduced to the slice

```
(1)  //  This program processes a list of integers input interactively during the
(2)  //  program run.  Count the number of non-negative integers in the
(3)  //  list, and find the maximum and minimum negative integer in the list.

(4)     #include <iostream.h>

(5)     int main ()
(6)     {
(7)        int current_value;
(8)        int current_maximum;
(9)        int current_minimum;
(10)       int non_negative_count;
(11)       int lcv;
(12)       int size;
(13)       int integer_array[100];

(14)       size = 0;
(15)       cout  << “Input an integer or -999999 to stop\n”;
(16)       cin >> current_value;
(17)       while (current_value != -999999)
(18)       {
(19)          integer_array[size] = current_value;
(20)          size++;
(21)          cout << “Input an integer or -999999 to stop\n”;
(22)          cin >> current_value;
(23)       }
(24)       current_minimum = 0;
(25)       current_maximum = 0;
(26)       non_negative_count = 0;
(27)       for (lcv = 0; lcv < size, lcv++)
(28)       {
(29)          if   (integer_array[lcv] < 0)
(30)          {
(31)            if  (integer_array[lcv] < current_minimum)
(32)                current_minimum = integer_array[lcv];
(33)            if  (integer_array[lcv] > current_maximum)
(34)                current_maximum = integer_array[lcv];
(35)          }
(36)          else
(37)             non_negative_count++;
(38)       }
(39)       cout  << endl  <<endl  <<endl;
(40)       cout  << “The number of non-negative numbers in the list was “
                 << non_negative_count << endl;
(41)       cout   << “The maximum negative number in the list was “
                 << current_maximum << endl;
(42)       cout  << “The minimum negative number in the list was “
                 << current_minimum << endl;
(43)    }
```

Figure 1. A buggy C++ program (fault area: line 25)

Slice	Statements, by number, in slice
Slice on Statement: #40 Variable: `non_negative_count`	4,5,6,7,**10**,11,12,13,14,15,16,17,18,19,20,21,22 23,**26**,27,28,29, **36**,**37**,38,39,**40**,43
Slice on Statement: #41 Variable: `current_maximum`	4,5,6,7,**8**,11,12,13,14,15,16,17,18,19,20,21,22,23 **25**,27,28,29,**30**,**33**,**34**,**35**,38,39,**41**,43
Slice on Statement: #42 Variable: `current_minimum`	4,5,6,7,**9**,11,12,13,14,15,16,17,18,19,20,21,22,23 **24**,27,28,29,**30**,**31**,**32**,**35**,38,39,**42**,43

Figure 2: Three static slices from the buggy code of Figure 1

on `current_maximum` at statement #41 (the second slice of Figure 2). This decreases the size of the program fault area by ten statements or 25%.

Knowing which output statements produce correct values can also help in fault-area reduction. A program *dice*, first defined by Weiser and Lyle [9], is a slice on one set of program variables at a statement minus a slice on a second disjoint set of variables at the same statement. If one is willing to assume correct output implies correct code[1], slicing a program on incorrect output variables and then dicing on correct output variables can reduce the program fault area even more than slicing on incorrect output variables alone does. In the above example, slicing on `current_maximum` and then dicing on {`current_min, positive_count`} gives a program fault area of five statements, {8,25,33,34,41}. This is a decrease in size of 87.5% from the original fault area.

Slicing and dicing provide a useful fault localization method. But do they also lend themselves to increased understanding of program code? The purpose of the exploratory study described in this paper is to show that a strong relationship exists between debugging with slicing and program understanding.

Since the concept of slicing was first formalized in the early 1980's, it has been the focus of much research. However, the emphasis of this research has been tool building and the issues and problems related to this activity. Applications of slicing and the advantages of using slicing have been largely neglected. This becomes unmistakably evident when one reviews any of the reference lists on slicing. (Note: Several of these lists can be accessed through the *Algorithmic and Automatic Debugging Home Page* [1].) For example, a total of only three of the 111 articles appearing in the bibliography of slicing maintained by Krinke [4] are concerned with issues of slicing applications or the advantages of using slicing. The same is true for the reference list of slicing articles maintained by Lyle [5]. Here applications and advantages again contains only the same three references. One of these references is to Weiser's dissertation [6]. One documents with empirical data that experts slice when debugging [8]. The third shows, using empirical data, that dicing when debugging saves time [9].

Clearly issues related to applications of slicing and the advantages of using slicing are an important piece in the overall slicing picture. Despite this, the amount of research devoted to these areas is small. Further investigation in these areas is needed to lend better understanding to the overall picture.

Below we explore the relationship between slicing while debugging and code understanding. We find that debuggers who use slicing have a better understanding of the program code after debugging than those who did not use slicing.

2. Experimental method

An experiment was conducted to test the hypothesis that slicing while debugging increases program understanding. For the experiment, subjects were first asked to debug a program, recording as they went the program statements they examined for correctness. As a follow-up activity, subjects were asked to construct from the program code the minimal subprogram that produced the incorrect output. Besides the statement list generated during debugging and the subprogram constructed in the follow-up task, start and stop times were recorded for both the debugging and reconstruction activities of the experiment. The experiment data was used to investigate the hypothesis that program debuggers who slice while debugging code gain better understanding of the program code than

1. There are exceptions to this assumption. For example, if an assignment statement contains the expression `a+b` where it should contain `a*b`, no program failure will occur if the code is tested with the value `2` assigned to both `a` and `b`. However, it is hoped that debuggers base their conclusions that a statement produces correct output on comprehensive suites of tests rather than on single tests, which makes the above type of exception less likely to happen.

those program debuggers who do not slice during debugging.

2.1 Subjects

The subjects were seventeen senior computer science majors at a small liberal-arts college in the Southeastern United States. Each volunteered for the experiment. At the time of the experiment all subjects had completed at least three computer science courses whose major emphasis was programming. All had programmed using Pascal in these courses. Also at the time of the experiment, each subject had completed between three and seven other computer science courses. None of the subjects had been exposed to the concepts of slicing or dicing. All participants received extra credit in a senior seminar course for their participation in the experiment.

2.2 Procedures

Each subject participated in a single, one-on-one session with the experimenter. There were no time limits imposed on any part of the experiment. The sessions ran anywhere from forty minutes to two and a half hours. The experimenter was available to the subject throughout the entire session. Each session consisted of three phases:

Phase I:	Training and practice
Phase II:	Program debugging
Phase III:	Subprogram construction

Phase I: Each subject began by reading the experiment instructions. The experimenter then reviewed orally with the subject the instructions and answered any questions the subject had regarding the instructions. Next subjects were given the task of debugging a practice program, recording the statements they examined for correctness. While practicing, subjects were encouraged to ask the experimenter regarding any experiment procedures that were unclear to them. Also, the experimenter randomly prompted subjects to be sure they were making a complete record of their debugging activities.

Phase II: Subjects were given the task of debugging a short program, recording as they went the statements they examined for correctness. Before they began debugging, subjects were given a sample input data file and shown both the output generated and the output expected when the program ran using the input data file. Subjects were not told how many or what type of faults the program contained. Each subject was provided a quiet space in which to work. This space included access to a familiar, on-line environment for Pascal programming.

Phase III: The program debugged in Phase II produced sixteen output values. For phase III, subjects were given the task of constructing from the Phase II program a subprogram that calculated and printed only the two output values of the Phase II program that were incorrect. Subjects were not expected to write-out the code for the subprogram, but rather were given the option of circling, underlining, or highlighting statements on a hardcopy listing of the experiment program code. Although experiment instructions read during Phase I informed the subjects there would be a follow-up task to the debugging task, they were not informed what this task would be until Phase III of the experiment began.

At the end of a subject's session, the subject was urged not to discuss the experiment materials or tasks with the other subjects.

2.3 Materials

For maximum control of slice sizes and intersections, the programs used in this experiment were written by the experimenter. The application domain of the program was chosen because it required minimal background knowledge.

Because all subjects had prior experience coding in Pascal, experiment programs were written in Pascal. Subjects did not have access to the program before their sessions. Two programs were used in the experiment, a practice program and the experiment program. Both programs were written in structured, indented style with no documentation but with descriptive variable names. Neither program contained any subprograms. The practice program was a short, 25-line program that calculated the average and standard deviation of a list of integers read from a file. The fault area of the program could be reduced to the single statement:

```
sum := sum + lcv;
```

with possible correction:

```
sum := sum + list[lcv];
```

The statement was embedded in a single `for` loop.

The experiment program was about 200 lines of Pascal that calculated a number of descriptive metrics on a file of text read by the program. These metrics included: counting blanks and non-blank characters, vowels, double blanks, double vowels, words, lines, blank lines, the maximum length word, the minimum length word, the maximum number of words per line, the minimum number of words per line, the maximum number of words per sentence and the minimum number of words per sentence. The fault area of this program could be reduced to the single statement:

```
persentence := 0;
```

with possible correction:

```
persentence := 1;
```

The statement was embedded in the fourth of four nested `if` statements.

To facilitate statement referencing, hardcopy listings of the program code with a line number attached to each statement were given to the subjects.

In addition to the listings of the two program codes, materials provided the subjects included:

Phase I:

1. Experiment instructions.
2. A copy of the mathematical formulas for average and standard deviation.

Phase II:

1. Access to the school's VAX system. The system has a platform for compiling, linking, and running Pascal programs. All subjects were familiar with this environment.
2. A hardcopy listing of an input file that could be used by the experiment program.
3. A print-out of the output produced by running the experiment program with the text file described in 2) as input.
4. A print-out of the correct output that should be produced by running the experiment program with the text file of 2).
5. A file containing the experiment program.
6. A file containing a copy of the text file of 2)

Phase III:

1. A new listing of the experiment program.

2.4 Hypothesis and Variables

The experiment on debugging and slicing tested the hypothesis: *Slicers have a better understanding of program code after debugging than do non-slicers.*

The independent variable was whether the subject was a slicer or a non-slicer. Slicers were determined from the data collected in Phase II of the experiment. All debuggers who stayed within the slice of the incorrect output variable were considered slicers; everyone else was considered a non-slicer.

The dependent variable in the experiment was understanding. Understanding was measured by the accuracy of program reconstruction during Phase III of the experiment.

2.5 Data analysis

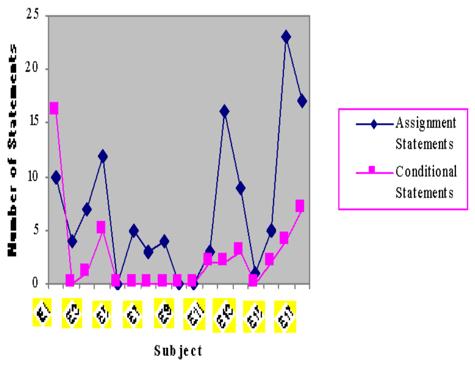
Phase II: Due to the small sample sizes of the slicer and non-slicer groups and the moderately skewed distribution of the test variables in Phase II, non-parametric statistical analyses were appropriate for comparing data collected from this phase of the experiment. The Mann-Whitney test was chosen.

Phase III: In Phase III of the experiment, distributions of the test variables was approximately normal, thus parametric statistical analyses were appropriate for comparing data collected from this phase of the experiment. The two-tailed t-test was chosen.

3. Results

During Phase II of the experiment subjects recorded the statements they examined for correctness and their start and stop times. Figure 3 shows the number of assignment and conditional statements outside the faulty slice each subject examined during debugging. Those subjects who examined one or no statements outside the slice were classified as slicers. All others were classified as non-slicers. Four of the subjects were found to be slicers while the remaining thirteen subjects were found to be non-slicers.

Figure 3: Statements Examind Outside the Slice during Debugging



Not surprisingly, slicers differed significantly from non-slicers in the number of statements they examined outside the faulty slice. One can also compare separately the number of assignment statements and the number of control statements each group looked at outside the slice. Here too the differences were significant.

- On the average slicers examined zero statements outside the slice while non-slicers examined twelve statements. This is a statistically significant difference of $p = .001$.

- On the average slicers examined zero conditional statements outside the slice while non-slicers examined three conditional statements. This is a statistically significant difference of p = .045.
- On the average slicers examined zero assignment statements outside the slice while non-slicers examined nine assignment statements. This is a statistically significant difference of p = .001.

Slicers and non-slicers also differed significantly on the number of statements they examined inside the faulty slice. Surprisingly non-slicers examined more statements of the faulty slice than did slicers.

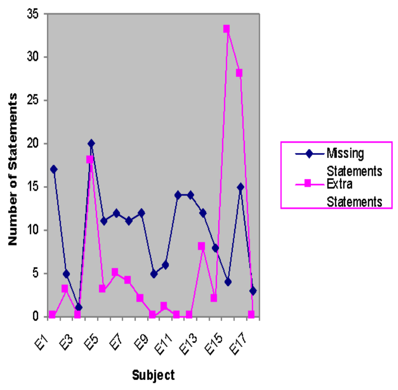
- On the average slicers examined nine statements inside the slice while non-slicers examined fourteen statements. This is a statistically significant difference of p = .0025.

The average time the slicers took to debug code was significantly less than the time taken by non-slicers.

- On the average slicers took fifteen minutes to debug the program while non-slicers took thirty-three minutes. This is a statistically significant difference of p = .045.

During Phase III of the experiment each subject underlined or highlighted or circled those lines of the original code that were needed to produce a subprogram that would calculate the maximum and minimum words per sentence of the text being analyzed in the faulty program of Phase II. Figure 4 shows the number of statements each subject did not include in the subprogram as well as the number of extra statements included.

Figure 4: Reconstruction Data



Slicers differed significantly from non-slicers in their accuracy during the reconstruction task of the experiment. Inaccuracy was based on both those statements that were missing and those statements that were unnecessary. Again we compare total statements and assignment and control statements separately.

- On the average slicers erred on nine statements in constructing the subprogram while non-slicers erred on twenty statements. This is a statistically significant difference of p = .02.
- On the average slicers erred on six assignment statements in constructing the subprogram while non-slicers erred on thirteen. This is a statistically significant difference of p = .03.
- On the average slicers erred on four control statements in constructing the subprogram while non-slicers erred on eight. This is a statistically significant difference of p = .02.

Subjects who were more accurate in their reconstructions took more time to complete Phase III of the experiment than those subjects who had a large number of inaccuracies in their reconstructions.

- On the average slicers took twenty-four minutes to construct the subprogram of Phase III while non-slicers on the average took eleven minutes. This is not a statistically significant different (p = .16)

4. Discussion

The experiment tested the hypothesis that slicers have a better understanding of program code after debugging than non-slicers, where understanding was measured through accuracy during reconstruction. The data gathered from the experiment supports this hypothesis. After debugging a program slicers were significantly more accurate in constructing a subprogram that isolated the faulty portion of the program than non-slicers were. This leads us to conclude that a relationship between slicing while debugging and program understanding exists.

The experiment did not investigate the question of causality. Further experimentation is needed before one can decide if the relationship between slicing while debugging and program understanding is caused by slicing or some other influence. However, the fact that the non-slicers examined significantly more of the faulty portion of the code during debugging than slicers did yet still showed significantly less understanding of the code at the conclusion of the debugging process indicates that such an investigation would be of value.

At the onset of the experiment, it was believed that reconstruction time could also be used to gauge understanding. Only after the data had been collected did it become apparent that a raw measure of time is inadequate in judging reconstruction time. A more meaningful time measure accounts for the time it takes to correct the inaccuracies made during reconstruction. The following formula is one way of doing this.

```
time = reconstruction_time +
    (# of extra_statements) *
        (time to remove a statement) +
```

Table 1: Three-way partition of subjects into dicers, non-dicing slicers, and non-slicers

	Dicers	Slicers	Non-Slicers
Average number of statements outside slice/dice examined while debugging	0	0	17
Average number of statements outside dice examined while debugging	0	3	21
Average number of inaccuracies during reconstruction phase	6	12	20

```
(# of missing_statements) *
    (time to add a statement)
```

There are several ways to estimate remove time and add time. However, it is hard to defend any such estimate as accurate. Unanswerable questions such as: did it take equal time to analyze each statement, did it take the same amount of time to *undo* as it did to *do*, and what to do about new errors all lead the authors to believe no valid conclusions can be drawn from the reconstruction time data collected.

In the following discussion we use the classification of novice (undergraduates), intermediate (graduate students or beginning professionals), and expert programmers (professionals with experience) suggested in *Empirical Studies of Programmers: Fourth Workshop* [3]. Weiser showed [8] that expert programmers use slicing while debugging. The data gathered in Phase II of the experiment discussed in this paper shows the opposite to be true with novice programmers. Of the seventeen subjects in our experiment only four used slicing. This is less than one-fourth of the sample size. Yet if slicing improves understanding, slicing may be a valuable tool for novice programmers. However, novice programmers will have to be taught slicing.

Of the four subjects classified as slicers in the experiment two of these are dicers. That is, two did not examine any statements outside the faulty dice. Table 1 shows statistics calculated with a separation of slicers who are dicers from slicers who are not dicers. Because of the small number of subjects in the two categories we do not generalize any of the results.

The conclusions mentioned above give rise to several interesting questions, each worthy of further investigation. The most obvious question is:

- Does debugging cause understanding?

But there are several others that merit further study too including:

- Does teaching novice programmers about slicing improve their debugging skills?
- Does dicing improve code understanding more than slicing during debugging?

The last two of these questions have ramifications in both computer science education and automated debugging tools. A strategy shown to provide both quick reduction of the code-fault area and better understanding of the code at the end of the debugging process is a strategy worth teaching the novice programmer. It also has value as part of an automated debugging tool.

Researchers are always searching for effective experimental methods that are non-invasive, easy to learn and which do not alter a subject's normal behavior. The method of recording statements as they are evaluated for correctness, used in this experiment, proved itself to be a method that meets all three of these criteria. Observation during the training sessions showed that recording statements does not change the subjects natural patterns of debugging. Also the training period needed to teach subjects how to record which statements they are evaluating for correctness was short yet effective. The time involved in training was about 20 minutes. Lastly the method is non-invasive. Subjects do not have to be observed while debugging nor prompted in any way.

References

[1] "Algorithmic and Automatic Debugging Home Page." http:// www.cs.nmsu.edu/~mikau/aadebug.html.

[2] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, New York, 1983.

[3] Jurgen Koenemann-Belliveau, Thomas G. Moher, and Scott P. Robertson (Eds.). *Empirical Studies of Programmers: Fourth Workshop*. Ablex.

[4] Krinke. "Bibliography." http:// www.cs.tu-bs.de/~krinke/ Slicing/slicing.html.

[5] James Lyle. "References to Program Slicing." http:// hissa.ncsl.nist.gov /~jimmy/refs.html.

[6] Mark Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*. Ph.D. Thesis, The University of Michigan, Ann Arbor, Michigan, 1979.

[7] Mark Weiser. “Program Slicing.” *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, California, March, 1981.

[8] Mark Weiser. “Programmers use slices when debugging.” *Communications of the ACM*, 25(7):446-452, 1982.

[9] Mark Weiser and James Lyle. “Experiments on Slicing-Based Debugging Aids.” *Empirical Studies of Programmers*, pp. 187-197, Ablex, 1986.