An Empirical Study of the Effect of Semantic Differences on Programmer Comprehension

David Binkley[†]
Loyola College in Maryland
binkley@cs.loyola.edu

Abstract

Software engineers face a wide range of difficult tasks. Understanding semantic relationships, such as the semantic differences between two programs, should aid a software engineer address many of these tasks. A series of experiments was conducted with an implementation of a semantic differencing algorithm for the C language. Sixty-three subjects participated in two controlled experiments. There is evidence that the experimental group, which had access to semantic differences, performed significantly faster ($p = 0.023$) and more accurately ($p = 0.047$) than the control group. The study provides empirical support to the assertion that semantic information assists program comprehension.

Key words: semantic differencing, empirical validation, program comprehension, program slicing

1. Introduction

It seems reasonable that having access to the semantic effects of a proposed change would aid software engineers, maintainers, and testers. This paper presents results from a study designed to formally test this belief. The particular semantic information studied is the semantic difference between two programs. A controlled experiment was conducted to determine if access to semantic differences is an effective aid to an engineer.

Semantic differences are useful in a wide variety of tasks. Examples include program integration [3], regression testing [4], and, the one studied herein, debugging. Debugging can be broken down into three phases: identifying the existence of an *error* (testing), finding the underlying *fault* (the cause of the error), and fixing the fault. This study deal with the latter two phases. The results demonstrate that access to semantic differences improves the fault locating and correcting ability of an engineer, and consequently, improve the debugging process.

Computing semantic differences is a difficult task. Few algorithms exist [1, 9]. An implementation of one of these is used in the experiment. This implementation takes as input two C programs `old` and `new`, and outputs a third C program `differences`. The significance of `differences` is that it captures the semantic (rather than syntactic) differences between `old` and `new` [4].

The implementation is used in two experiments. The first includes two debugging tasks that require understanding a program well enough to identify a fault. The second experiment involves the detection of (unwanted) ripple effects. These experiments seek to discover if access to semantic differences aids an engineer. Positive results should facilitate the transfer of semantics based tools from academia to industry.

It is worth stressing that the algorithm is semantics based: it makes use of (safe approximations to) program semantics to capture the *computations* of `new` that differ from the *computations* of `old` [4]. The capturing of semantic differences is based on the formal semantics of `old`, `new`, and `differences`. It extends to runtime behavior under a deterministic computational model, where each time a program is executed on the same input, it produces the same output. In practice, this model may

[†]supported by National Science Foundation grant CCR-9803665.

be violated when, for example, the location at which a program is loaded affects its behavior [4].

The remainder of this paper is organized as follows: Section 2 presents some background material. Results from the two experiments, which make up the main technical contributions, appear in Section 3. Finally, Section 4 discusses related work, and Section 5 summarizes the paper.

2. Background

This section provides background on the algorithm used to compute `differences` [4]. The algorithm is based on the system dependence graph (SDG), which is a collection of procedure dependence graphs (PDGs) connected by interprocedural control-dependence and data-dependence edges [3, 10]. (The example later in the section includes a program and its SDG.)

Each PDG contains vertices, which represent the components (statements and predicates) of a procedure, and edges, which represent the control and data dependences between components. The dependences in a PDG are safe approximations to the semantic dependences found in the program [13], which are generally not computable and therefore must be (safely) approximated.

Backward and forward interprocedural slices of the SDG [10] are used to compute `differences` [3, 4]. These slices are computed using a two-pass closure algorithm over the edges of the SDG. Given a program `old` and a program `new`, created by modifying `old`, the algorithm computes the following sets:

Directly Affected Points

The set of vertices *directly affected points* is a safe approximation to the components from `new` directly influenced by the change to `old` [3]. It is computed by comparing the SDGs of `old` and `new` [11].

Affected Points

The set of vertices *affected points* is a safe approximation to the components whose behavior (*interpretations* [13]) may differ in `new` and `old`. It is computed using forward slicing [3].

`differences`

`Differences` is an executable program that captures the computation of the affected points. It is computed using backward slicing [2].

Example. The program in Figure 1 extends `old`, which sums the first 10 numbers, to also compute the product of these numbers. The computation of `differences` proceeds as follows: The only directly affected points are the vertices that represent new statements. These are underlined in the figure. In addition to the directly affected points, the set of affected points includes those vertices whose computation is affected by a directly affected point. In Figure 1, these are the vertices that make up procedure *Add*, whose behavior is different (completely new) when called from *Multiply*. The program `differences` includes those statements necessary to support the computation of the affected points. In the example, this includes the computation of `i`, which is not affected by the change, but is required to capture the computation of `prod` in an executable program. Program

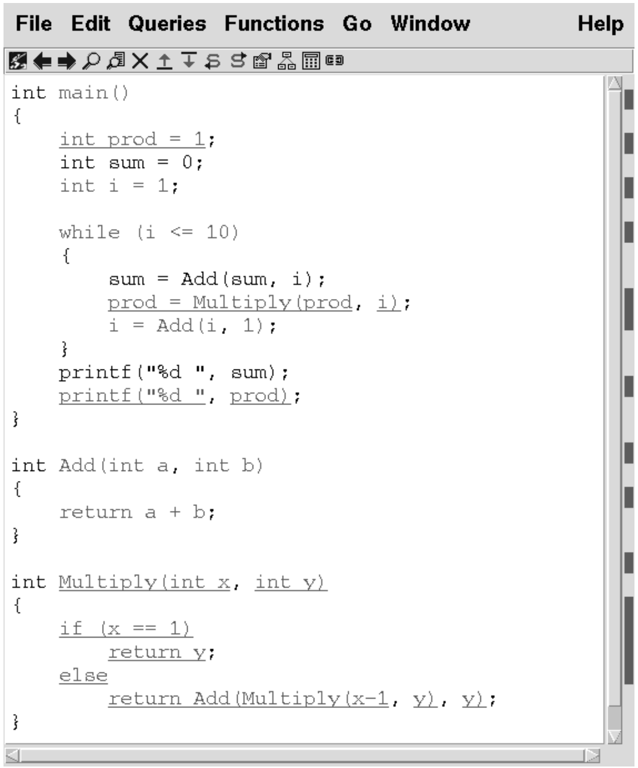

Figure 1. The product program. `New` **was created by enhancing** `old` **with the computation of** `prod`**. (Function** `Multiply` **assumes its first argument is positive.)**

`differences` is highlighted in Figure 1 by CodeSurfer [8] (the program analysis tool upon which the implementation is based). CodeSurfer highlights in red (which appears gray in the figure) the relevant statements. Figure 2 shows the SDG for this example.

The experiments make use of an optimized version of the algorithm that is more efficient than producing `new` and then computing `differences`. Instead, `differences` is computed with respect to a hypothetical `new` program. In essence an engineer is asking the question "if I change statement *S* then what statements would appear in `differences`?" Thus, `new` need not be materialized.

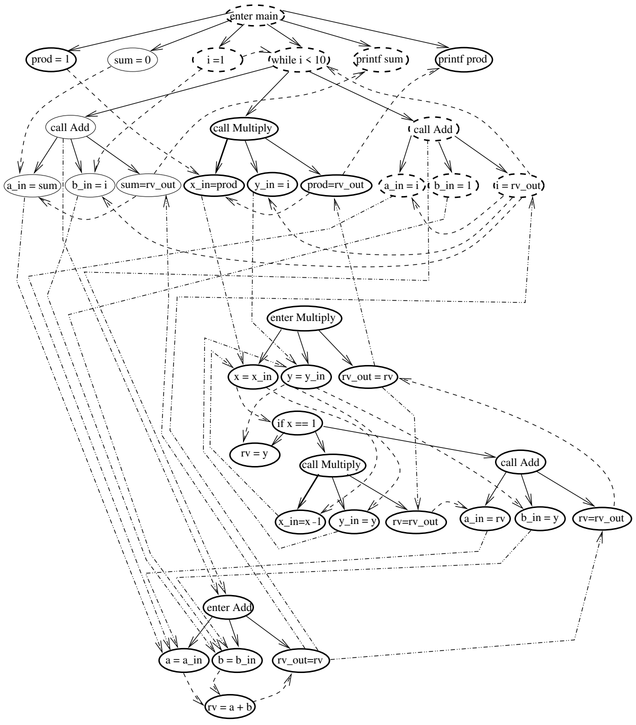

Figure 2. The product SDG post analysis. Affected points are highlighted in bold, (all but those in procedure `Add` are directly affected points). Other components of `differences` are highlighted in bold dashed lines. Non-bold components (representing the computation of `sum`) are not in `differences`.

3. The Experiments

Experiments were designed and administered to assess the value of semantic difference information in several software engineering tasks that would presumably benefit from increased programmer comprehension. This section first discusses the experimental setup and design. It then considers quantitative results, and finally, data from subjective questions, which provide a context for the numerical data.

3.1. Experimental setup and design

The experiment was designed to test the general hypothesis that access to semantic differences aids in program comprehension. Specifically, two hypotheses are under test:

(1) access to semantic differences makes programmers more *efficient* (faster) at tasks that require program comprehension, and

(2) access to semantic differences makes programmers more *accurate* at tasks that require program comprehension.

Sixty-three subjects in eight groups took part in the study. The subjects included undergraduate students halfway through the second, third, and fourth years of study at Loyola College. The program at Loyola College is ABET accredited; thus, the subjects has seen standard computer science instruction through their corresponding year of study. The subjects has competency in C, the language used in the programs, appropriate to their year of study.

There was some overlap in the three courses that provided the subjects; thus, two test instruments were developed (one for first time subjects and one for second time subjects). In all, four experiments were conducted: three with first time subjects and one with second time subjects.

Each experiment was administered in a fifty minute class period. It began with a ten minute lecture that overviewed the approach, and included two worked examples. The same person gave this lecture to all eight groups. Although the lecture was not given to all groups as the same time, the same set of notes was followed to avoid biasing any particular group.

After the introductory lecture, the programs and problems were handed out. The remaining forty minutes were alloted to complete the problems.

First time subjects were presented with two problems: a small 50 line C version of the `word count` program (which counts the number of characters, words, and lines in its input), and the CP/M disk utility `wash`, which contains just over 500 lines of C code. Second time subjects were given a modification of a 200 line C program that computes parking fees.

Following each program was a questionnaire, which asked participants about the program. Figure 3 shows some sample questions. The answer sheet also asked subjects to rate their confidence in their answers. Finally, subjects were asked to reconstruct the program.

The rationale behind the questions is to investigate several things. Questions A1 and A4 (see Figure 3) established the time taken to performed a task. Questions A2 and A3 were used to measure correctness. Finally, reconstruction questions were intended to determine if the experimental group subjects were more focused in their recall. If so, they should correctly identify relevant statements and mis-identified irrelevant statements.

Many of the questions allow rather "free form" answers; however, the scoring was "all or nothing" and responses were either clearly right or clearly wrong. The two exceptions to this are discussed later in this section. To avoid scoring bias the papers were graded blind and in a random order.

During each run of the experiment, subjects were divided into a control group and an experimental group. Group assignment was done by the course instructor to avoid an unequal distribution of talent, which would bias the results. The only difference between the two groups was that the experimental group had semantic differences highlighted in the programs they were given.

When comparing various groups, a variety of statistical tests are used. The null hypothesis in each case is $\mu_1 - \mu_2 = 0$. For timing questions, the alternative hypothesis is $\mu_1 - \mu_2 > 0$ (*i.e.*, the experimental group is faster); for the confidence questions, the alternative hypothesis is $\mu_1 - \mu_2 < 0$ (*i.e.*, the experimental group was more confident). In each case, resulting p-values less than 0.05 are interpreted as significant.

Several of the analyses assumes that the parent populations have normal deviations. This normality assumption was verified, using the Anderson-Darling test for normality, and was satisfied in all cases. Furthermore, most of the tests are known to be robust against non-normality of the data. The present analysis assumes that the standard deviations are unequal (heteroscedastic). Repeating the analysis assuming they are equal (homoscedastic), does not affect any of the conclusions. Numerically, for the significant results, the p-values change (some up and some down) by no more than 0.006.

3.2. Quantitative results

Overall, there is statistical evidence that access to semantic differences makes programmers faster and more accurate. For each of the three programs, the data related to the two hypotheses are considered below. Each program is followed by questions designed to test the focus of the subjects.

First Time Subjects

For first time subjects, Tables A and B contain averages of individual groups and various combined groups, respectively. Table C presents results from two-way ANOVA tests used to analyze some of this data. The data and results shown in these tables is now considered in greater detail.

First, Program 1 was a version of the `word count` program, which contained a fault in the initialization of the line count. The fault caused an error in the line count output. Program `differences`, computed from this output, essentially highlights the statements from the original program that make up a program that only counts lines.

A1) What time is it: ______________________

A2) Study the first program. What do you think the problem is with the program?

A3) Suggest a solution.

A4) What time is it: ______________________

Figure 3. Answer sheet excerpts.

For Program 1, the timing data related to Hypothesis I includes the time taken to study the program and then answer Questions A2 and A3 (see Figure 3). Two factors potentially influence the times: the group (treatment vs. control) and the class (under vs. upper). Overall, as seen in Table B, the average time of 7.25 minutes for the control group appears slower than the 5.17 minutes for the experimental group. Looked at by class, the average for the underclassmen of 6.49 minutes appears slower than the 5.93 minutes for the upperclassmen.

To analyze this numerical data for time, a two-way ANOVA was applied. In essence, this test considers three things: First, is their an interaction between the group and class-year factors; second, does the group have any effect on the result, and finally, does class-year have any effect on the result. The p-values for these three tests are given in Table C.

Table A — Individual Group Averages

Group:	underclassman		upperclassman	
	con	exp	con	exp
Program 1				
Time (minutes)	7.14	5.83	7.35	4.50
% Correct (A2)	57%	75%	79%	93%
% Correct (A3)	64%	75%	71%	93%
Program 2				
Time (minutes)	6.43	6.92	5.43	10.1
% Correct	0%	7.1%	8.3%	28.6%

con - control group under - underclassmen
exp - experimental group upper - upperclassmen

Table B — Combined Averages

Group:	con	exp	under	upper
Program 1				
Time (minutes)	7.25	5.17	6.49	5.93
% Correct (A2)	68%	85%	65%	86%
% Correct (A3)	68%	85%	69%	82%
Program 2				
Time (minutes)	5.93	9.49	6.67	7.75
% Correct	3.6%	19.2%	3.8%	17.9%

The first test shows that there is no evidence of interaction between the two factors with regard to the time for Program 1 ($p = 0.39$). Thus, the two factors can be considered separately. In support of Hypothesis I, the experimental group is significantly faster ($p = 0.023$).

One might expect that greater maturity of upperclassmen would lead to improved performance. Interestingly, there is no statistical evidence that class-year affects the time taken ($p = 0.53$). This indicates that the treatment and not maturity is causing the difference.

One additional consideration for the control groups is evidence that some of the control subjects (particularly the underclassmen) were giving up on finding a solution. This makes the control underclassmen's times artificially low, but raises the standard deviation (the experimental group's standard deviation is 2.45 compared to 3.82 for the control group). The potential impact of this "giving up" is discussed later in this section.

Next consider the correctness data related to Hypothesis II. This data comes from answers to Questions A2 and A3. The results for the problem identification (Question A2) and solution (Question A3) are the same. From the data in Table A, it appears that groups and class-year influence the percent correct. Overall, as shown in Table B, the experimental group gave a higher percentage of correct answers than the control group (85% versus 68%).

Despite the higher percent correct of the experimental group, there is no statistical evidence to support Hypotheses 2. Because the data is essentially binary, a χ^2 test was used. For both questions the experimental group's apparent better performance is not statistically significant ($\chi^2 = 2.07$, $p = 0.15$).

Table C — Two-way ANOVA p values

	interaction	group	class-year
Program 1			
Time (minutes)	0.39	0.023	0.53
A2 Confidence	0.69	0.99	0.004
A3 Confidence	0.77	0.69	0.003
Program 2			
Time (minutes)	0.041	0.013	0.28

In table B, most subjects who correctly answered Question A2, correctly answered Question A3. Two exceptions are a control underclassman who failed to articulate the problem, but was nonetheless able to provide a correction and a control upperclassman who identified the problem but did not provide a correct solution. When looked at by group these two cancel each other; when looked at by class-year they cause a slight difference in the percent correct.

In summary, for Program 1, the experimental group is significantly faster and appears to be giving more correct answers although there is a lack of statistical evidence for this second claim. One cause of this is the binary nature of the data, which leads to lower power of the statistical test.

In addition to the time and correctness questions, a final set of questions were designed to ascertain the focus of subjects. This techniques follows that used by Francel and Rugaber for measuring programmer comprehension [7]. Subjects were asked to identify the statements they considered during their initial study of the program and those relevant to the fault in the program after answering the questions (without looking back at the original code).

Over half (52%) of the statements considered by the experimental group were affected points. In contrast, over half (57%) of the statements considered by the control group were not in `differences` at all (let alone affected points). In the reconstruction, 51% of the control group subjects and 69% of the experimental group subjects identified the statements that were affected by the change. Thus, highlighting that semantic differences help focus subjects on the relevant parts of the code.

Upperclassmen did a better job of reconstruction than underclassmen. This pattern is observed in other domains. For example, a chess grand master can reconstruct a board position given only a glance at the board. The same is not true of less experienced players.

The final two questions designed to measure subject focus and comprehension were “What determines where a word starts or stops?” and ”In your own words describe what the program does.” Of the control subjects, 43% correctly answered the first question. Since the word computation was not related to the fault, this provides evidence of wasted effort. In contrast, only 11% of the experimental subjects correctly answered this question. Many of their incorrect answers clearly reflected intuitive guesses at how word identification might have been done. Thus, the experimental group is focused on the relevant code and ignoring irrelevant parts of the program. This provides indirect evidence in support of Hypotheses I and II as the experimental groups is more focused and thus should be more efficient.

For the second question (“In your own words describe what the program does”), 93% control group subjects gave the correct answer. The remaining two responded “?” and “it outputs letters, lines, and characters.” In contrast, only 62% experimental group subjects gave the correct answer. Incorrect responses included five identifying the counting of lines only, (which indicates the desired focus on the line count computation). Other responses included one lines and characters only, one lines and words only, and three lines, words, characters, and *sentences*. This again illustrates that the control groups are focusing where they need not.

Program 2, `wash`, is just over 500 lines of C code. It performs a collection of operations (copy, delete, rename, etc.) on a list of files. The copy file function exhibits an error: the target file is always empty. Subjects first studied the code before being asked to identify the fault causing this error and suggest a solution. Finally, they were given a collection of function names and asked to identify, without looking back at the code, which were included in the program.

The fault occurs in the function `moveFileData`, which is responsible for copying data from one file to another. The function includes the statement

```
while((cnt=read(tofd, buf, len))>0)
```

instead of the statement

```
while((cnt=read(fromfd, buf, len))>0)
```

Here `differences` was computed from the results of a failed test run that uncovered an error when the file copy operation was tested. Program `differences` was computed by using as the directly affected points, the body of the if statement executed when the user selects the copy file option from the main menu.

Tables A and B include the average times and percents correct for this program. The first thing to note about the time data is that the experimental groups, in particular the experimental upperclassmen, appear to actually take longer. This is the opposite of the expected result. The correctness related data follow the expected trend with experience and the treatment appearing to improve the percent correct.

The results of the two-way ANOVA related to Hypothesis I (see Table C) show significant evidence of interaction between the group and class-year factors; thus, it is impossible to consider the two factors independently. Instead, Tukey's pairwise comparison among means was applied. This test shows that the mean for the experimental upperclassmen is not statistically different from that of the experimental underclassman (Tukey simultaneous pairwise comparison adjusted p-score 0.14). Nor is it different from the upperclassmen control, but just barely (adjusted p-score 0.051). Finally, it is statistically different from the control underclassmen (adjusted p-score 0.0075). All other pairwise comparisons showed no significant difference. Thus, the experimental upperclassmen took significantly *longer* than the control underclassmen. It is impossible to isolate the cause of this effect.

This interaction and the inversion from the expected completion time, are potentially explained by what might be called the "throwing in the towel" effect. Given the number of incorrect (often ridicules) answers provided by the control group, it is likely that many subjects from this group were giving up on the task ("throwing in the towel"). Further evidence of this is the number of control group times of three minutes or less, which is clearly insufficient time to understand 500 lines of code. Subjectively, this effect was obvious in the subjects demeanor and in their answers to subjective questions about the experience. Finally, it should be noted that participation is the experiment is voluntary and human subject rules forbid performance based rewards being offered as an incentive.

One way of eliminating the "throwing in the towel" effect would be to compare only the times for correct answers. However, since only one control group subject gave the correct answer, this is not feasible.

Next, consider Hypothesis II. Of the 54 subjects, seven correctly identified the fault: one control group subject (an upperclassman) and six experimental group subjects (including one underclassman). In addition, two experimental group subjects came "close." One responded "I think the problem may be in the function `moveFileData` in the while loop" and another "moveFileData does not work. It never writes anything to the file. $\cdots$ Fix the while loop so it executes." (While this second answer appears to represent a correct solution, only responses that explicitly mentioned `tofd` were counted as correct. These are the only such borderline answers.)

The results related to Hypothesis II indicate that the experimental group performed significantly better. Here the χ^2 test reports a p score of 0.033. However, since the sample size is small and the assumptions of the χ^2 test are violated, Fisher's exact test was also used. The result is statistically significant ($p = 0.047$). Given they binary nature of the data this results indicates a strong pattern.

In summary, for Program 2, the experimental group is more accurate, but can not be shown to be faster; thus, there is support of Hypothesis II, but not Hypothesis I.

As with Program 1, questions designed to determine if subjects were focusing on relevant or irrelevant code followed Program 2. In this case a collection of function names was included. Two thirds of these functions actually appear in the program. Two of these are of particular interest: `CopyFile` and `moveFileData.` Given the description of the bug, the subjects were expected to study, and thus correctly identify, the `CopyFile` function. This in fact happened, as 93% of the control group and 92% of the experimental group correctly identified this function as occurring in the program.

The other function `moveFileData` was called from several locations and it is not immediately obvious that it is relevant to the fault. It does appear in `differences`. The study of this function by more experimental subjects than control subjects was clear, as 85% of the experimental group subjects and only 39% of the control group subjects correctly identified this function.

Finally, consider two control group's responses to the questions "What do you think the problem is?" and "Suggest a solution." These responses were "Where did [function] moveFileData come from?" and "Find the code that is supposed to write the data to the new file and find [the] errors." While the first of these could be resolved by a text editor, the second requires deeper analysis. In this case `differences` is that analysis; it includes precisely the code the programmer is looking for.

Overall for the first time subjects, there is statistical evidence that the experimental group is both faster and more accurate than the control group; thus, affirming both Hypotheses I and II. For Program 1, the problem was simple enough that both groups achieved high scores; however, the experimental group was faster. For Program 2, the problem was so difficult that control groups subjects appeared to be giving up on finding a solution; thus, there is an anomaly in the timing results. The increased difficulty did allow the experimental group to attain a statistically greater number of correct answers.

Second Time Subjects

Nine subjects participated in the experiment twice. The second time they were given a program and a proposed modification and asked if the modification correctly implemented a change in the program's specification. While it was not explicitly stated in the instructions, the problem was designed to test `differences` ability to assist in the identification of unwanted ripple effects, which the proposed modification introduced.

With only four control subjects and five experimental subjects, there are no statistically significant results from this second experiment. Informally, the following observations are of interest: first, one of the four control subjects and two of the five experimental subjects correctly discovered the (unwanted) ripple effect. Second, there was some evidence that the control group was again giving up on the problem as their average time of 8:15 was less than the experimental group's average time of 10:24.

3.3. Subjective questions

To get a feeling for the subject's opinion of their ability and performance, they were asked to rank their confidence in their answers on a scale from 0 (I guessed) to 4 (I'm sure). Table D summarizes the confidence responses for Questions A2 and A3 of Figure 3.

The tests for interaction, summarized in Table C, show no statistical evidence of any interaction between the two factors. Thus, the factors can be looked at independently. Unfortunately, there is no statistical support for the claim that the treatment group was more confident in their answers ($p = 0.99$ for Question A2 and $p = 0.69$ for Question A3). However, there is rather strong evidence that the upperclassmen were more confidant of their answers ($p = 0.004$ for Question A2 and $p = 0.003$ for Question A3).

Table D — Question A2 and A3 Confidence

	A2: problem identification			A3: problem solution		
Group	$\bar{x}$	s	n	$\bar{x}$	s	n
control	3.01	1.21	27	2.80	1.32	25
experimental	3.02	1.06	26	2.98	1.04	26

Had the upperclassmen performed better than the underclassmen, this confidence might have been warranted. However, there is no evidence that they provided more correct answers. For Question A2, upperclassmen gave 86% correct responses compared with 65% for the underclassmen; however this result is not statistically significant ($\chi^2 = 3.05$ $p = 0.076$). Nor is correctness for Question A3 ($\chi^2 = 1.23$ $p = 0.22$), where upperclassmen gave 82% correct responses compared with 69% for the underclassmen.

Subjects also rated their confidence when answering questions associated with Program 2. Unfortunately, do to an error in the survey instrument, the data are not meaningful. The error was that subjects were asked to rate their confidence in their answer and not their confidence in the *correctness* of their answer. Thus, when asked "what is the problem with the program?" several control group subjects responded "It does not work" and then gave their confidence as 4 (highest). Other similar non-answers were given with high confidence. While this invalidates the confidence data, it does appear to be further evidence that these subjects were giving up on finding a solution. Most such responses came from control group subjects; a few came from experimental group underclassmen.

4. Related work

Perhaps owing to its inherent complexities, semantic differencing is an under-studied area. This section briefly considers a complimentary experiment run using `differences` and then related slicing based work by Francel and Rugaber [7], and Gallagher and Lyle [12]. There is too wide a variety of work related to the applications of semantic differencing and program comprehension to do them justice here.

An implementation of the algorithm described in Section 2 was used to study the size of `differences` for a collection of ten mid-sized programs (ranging in size from 3,000 to 22,000 lines of code) [5]. The study assessed the size reduction obtained computing `differences` assuming that each statement in turn was the source of a change. The average reduction was 37.7%. The average size of `differences` was 7525 LOC, which required an average of 1.25 seconds to compute. Individually, Table E shows statistics related to the size reduction obtained for each of the programs.

Table E — Size Reductions				
program	LOC	SDG size (in SDG vertices)	`differences` mean size (in SDG vertices)	mean percent reduction
a2s	22,068	57,922	30,161	47.93%
acct	10,182	110,860	29,830	73.09%
barcode	5,848	195,62	13,209	32.48%
bc	8,364	70,502	64,909	7.93%
copia	4,488	46,130	29,404	36.26%
ed	13,578	98,479	79,823	18.94%
flex	21,543	53,418	29,660	44.48%
oracolo2	14,382	92,200	65,918	28.51%
prepro	14,330	95,757	68,321	28.65%
termutils	7,006	10,161	4,192	58.74%
average	12,079	65,499	41,543	37.70%

The average of 37.7% is an improvement over the 26% average reduction obtained with a collection of smaller programs [5]. Such an improvement is consistent with the notion that larger programs contain more separable pieces. A similar kind of improvement has been observed in the hardware domain [6].

Francel and Rugaber conducted a comprehension experiment similar to that described in Section 3. They considered the relationship of slicing and debugging to program understanding. Using the accuracy of reconstruction as a measure of understanding, they concluded that slicers (those who were determined to be computing slices while debugging) had better understanding of the code after debugging. The study from Section 3 provides additional evidence to the claim made by Francel and Rugaber. It also helps to dispell the concern that subjects who used slices in the Francel and Rugaber study were simply better students, and thus, did a better job of reconstruction.

Finally, the maintenance technique of Gallagher and Lyle uses slicing to decompose a program into changeable and unchangeable parts. By forbidding modifications in the unchangeable part, unwanted ripple effects are avoided. In contrast, semantic differencing, as used in Section 3, only highlights potential ripple effects. However, it does not restrict allowable modifications. This can be viewed as a tradeoff between living within the confines of a tool that restricts the introduction of unwanted ripple effects and running the risk of missing some.

5. Summary

This paper describes experiments in which semantic differences were found to aid programmers in tasks that require program comprehension. This human factors study complements a previous study on the size reduction obtained by `differences` [5]. Together, these studies provide evidence that semantic differences can serve as an aid to an engineer.

In any experiment, it is important to consider so called “threats to validity.” Several such factors may affect the validity of these results. First, there is the issue of external validity; in other words, will these results transfer to other organizations or settings.

Second, there are potential skewing influences on the results produced by the rules that govern the use of human subjects. In particular, participation must be voluntary and no performance based rewards may be offered. This is expected to have the following impact: the control group is more prone to give up and not really apply themselves while the experimental group may work harder as they know they are the "chosen" group. Further studies in different environments will help alleviate these concerns.

Finally, the second group of upperclassmen participated in the experiment three days after the other subjects. Thus, it is possible that some of them gained advanced knowledge of the programs and questions.

Debugging can be broken down into three phases: discovering an error (*e.g.*, an incorrect output), then finding the cause of the error (*i.e.*, the fault), and finally fixing the fault. This study deals with the later two phases. In particular, it shows that access to semantic differences, improves comprehension, and thus allows subjects to better find and fix faults. This leads to increased correctness, speed, and the identification of unwanted ripple effects.

In prior work, `differences` has been shown to aid in finding errors during regression testing [4]. Looking forward, the next step in this research is to investigate if `differences` can help testers locate faults in general and not just during regression testing.

A simplistic summary of the results reported in this paper is that showing a programmer less code makes him or her more effective. One might mistakenly look on this result as trivial. After all, showing a programmer less of the program should make him or her more effective. However, the real result of this experiment is that `differences` selects the correct code to show the programmer *automatically*.

Acknowledgments

GrammaTech Inc. (www.grammatech.com) provided *CodeSurfer*[tm] upon which the implementation is based. Chris Morrell was of great help in the statistical analysis of the data. Keith Gallagher's comments improved the exposition.

References

1. Binkley, D., "Using semantic differencing to reduce the cost of regression testing," pp. 41-50 in *Proceedings of the IEEE Conference on Software Maintenance,* (Orlando, FL, Nov, 1992), IEEE Computer Society, Washington, DC (1992).
2. Binkley, D., "Precise executable interprocedural slices," *ACM Letters on Programming Languages and Systems* **2**(1-4) pp. 31-45 (1993).
3. Binkley, D., Horwitz, S., and Reps, T., "Program Integration for Languages with Procedure Calls," *ACM Transactions on Software Engineering and Methodology* **4**(1) pp. 3-35 (January 1995).
4. Binkley, D., "Semantics guided regression test cost reduction," *IEEE Transactions on Software Engineering* **23**(8) pp. 489-516 IEEE Computer Society, (August 1997).
5. Binkley, D., Capellini, R., Raszewski, L., and Smith, C., "An implementation of and experiment with semantic differencing," *In Proceedings of the 2001 IEEE International Conference on Software Maintenance, (Florence, Italy)*, pp. 82-91 (November 2001).
6. Clarke, E.M., Fujita, M., Rajan, S.P., Reps, T., Shankar, S., and Teitelbaum, T., "Program slicing of hardware description languages," *Conference on Correct Hardware Design and Verification Methods*, pp. 298-312 (1999).
7. Francel, M. and Rugaber, S., "The relationship of slicing and debugging to program understanding," *Proceedings of the Seventh International Workshop on Program Comprehension*, pp. 106-113 (5-7 May 1999).
8. Grammatech Inc., *http://www.grammatech.com/ products/codesurfer/codesurfer.html*. 1999.
9. Horwitz, S., "Identifying the semantic and textual differences between two versions of a program," *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 20-22, 1990)*, ACM SIGPLAN Notices* **25**(6) pp. 234-245 (June 1989).
10. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems* **12**(1) pp. 26-60 (January 1990).
11. Horwitz, S. and Reps, T., "Efficient comparison of program slices," TR-982, Computer Sciences Department, University of Wisconsin, Madison, WI (December, 1990).
12. Lyle, J.R. and Gallagher, K.B., "A Program Decomposition Scheme with Applications to Software Modifications and Testing," *Proceedings of the 22nd Hawaii International Conference on Systems Sciencea (Volume II, software track)*, pp. 479-485 (January 1989).
13. Podgurski, A. and Clarke, L.A., "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on Software Engineering* **16**(9)(September 1990).