An Empirical Study of Amorphous Slicing as a Program Comprehension Support Tool

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

Mark Harman‡
Goldsmiths College University of London
m.harman@gold.ac.uk

L. Ross Raszewski†
Loyola College in Maryland
csmith@cs.loyola.edu

Christopher Smith†
Loyola College in Maryland
lraszews@cs.loyola.edu

Abstract

Amorphous program slicing relaxes the syntactic constraint of traditional slicing and can therefore produce considerably smaller slices. This simplification power can be used to answer questions a software engineer might have about a program by first augmenting the program to make the question explicit and then slicing out an answer. One benefit of this technique is that the answer is in the form of a program and thus, in a language that the software engineer understands well.

To test the usefulness of amorphous slicing in answering such questions, the question of array access safety is considered. A safety slice, an amorphous slice of an augmented program, is used to guide a software engineer to potential array bounds violations.

A series of experiments was conducted to determine whether the safety slice was an effective aid to an engineer. Seventy six subjects participated in the controlled experiments. For experiments involving novice programmers, the null hypothesis could not be rejected, and so it was not possible to conclude that amorphous slicing assisted such programmers. However, for more experienced groups, the experimental subjects (who were able to consult amorphous slices) significantly outperformed the control group. The study lends empirical support to the assertion that amorphous slicing assists program comprehension.

Keywords: slicing, empirical validation, amorphous slicing, arrays, program comprehension

1. Introduction

Program slicing is a decomposition technique that extracts from a program statements relevant to a particular computation. Informally, a slice provides the answer to the question "What program statements potentially affect the computation of variable v at statement s?" An observer viewing the execution of a program and its slice cannot determine which is which when attention is focused on statement s and variable v.

This definition places a semantic constraint on the slice; it requires the slice to have a certain behavior. In his original definition, Weiser also placed a syntactic constraint on a slice [14]: a slice is determined by deleting statements from a program. For debugging, the application Weiser was studying, this is a desirable property. However, for some applications, such as program comprehension, it can be overly restrictive.

If the syntactic requirement is relaxed, the result is an *amorphous program slice* [9]. An amorphous program slice has the same semantic constraint as a traditional slice, but a more relaxed syntactic constraint. As illustrated in Figure 1, this allows certain transformations to reduce the size of the slice.

Amorphous slicing has been applied to the static analysis of array access safety [9, 3]. Not surprisingly, like all interesting static analyses, it is impossible to fully automate the detection of array access violations. Therefore, some human comprehension effort will always be required. It has been claimed [3, 10, 9] that amorphous slicing can be used to aid comprehension, reducing human analytical effort. One reason for this approach is

† Supported by National Science Foundation grant CCR-9803665.
‡ Supported by Engineering and Physical Sciences Research Council grants GR/M58719 and GR/M78083.

Initial Program	Traditional Slice	Amorphous Slice
main()	*main*()	*main*()
{	{	{
$a = 1$	$a = 1$	
$b = a + 2$	$b = a + 2$	$b = 3$
$c = 5$		
}	}	}

Figure 1. A traditional and an amorphous slice of a program. The slice is taken with respect to b at the end of the program. Traditional slicing deletes the statement "$c = 5$." Amorphous slicing further reduces the program's size using constant propagation and constant folding.

that the results are reported as a program, not in some logic formalism; thus, facilitating understanding by the engineer. This paper reports the results of a set of experiments designed to test these claims.

The remainder of this paper is organized as follows: Section 2 briefly describes the algorithm used to compute the amorphous slices. Section 3 applies amorphous slicing to the problem of identifying array subscript violations, producing the *safety slice*. The results of experiments into the effectiveness of the safety slice, the main technical contribution of this work, are presented in Section 4. Finally, a summary appears in Section 5.

2. Background

This section describes the System Dependence Graph (SDG) [12] and an algorithm, based on the SDG, for computing amorphous slices [3, 9]. A program's SDG is a collection of vertices and edges. The vertices represent the components of the program (*e.g.*, assignment statements and predicates). The edges represent dependences between components.

There are two kinds of edges use in the algorithm: control-dependence edges and flow-dependence edges. A control-dependence edge, which is labeled either **true** or **false**, connects a vertex v to a vertex u if during execution, whenever the predicate represented by v is evaluated and its value matches the label on the edge to u, then the program component represented by u will eventually be executed provided the program terminates normally. A flow-dependence edge runs from a vertex that represents an assignment of a variable to a vertex that represents a use of the variable reached by that assignment.

Figure 2 presents an algorithm for computing amorphous slices. The algorithm has four steps: compute the SDG for the program, slice this SDG, transform the slice using a set of rewriting rules, and finally reconstitute a program from the transformed graph. After the SDG is built, the second step is a traditional slice. This slice removes irrelevant computations from the program.

The third step applies a collection of rewriting rules to the slice, many of which essentially perform standard compiler optimizations (*e.g.*, loop fusion [5]), to further simplify the graph [9]. Table 1 contains an example rule; Steps 2 and 3 are repeated until no further simplification results.

The fourth step produces a program from the rewritten SDG. To do so requires making the SDG *feasible* (a feasible SDG is the SDG of some program) [3]. The graph is then reconstituted into a program [1, 4, 11].

3. The Safety Slice

This section defines the *safety slice* and briefly discusses its computation [3, 9]. Slicing is not directly applicable to the problem of identifying array access violations because

```
function AmorphousSlice(P, S)
declare
   SlicedP, P: programs
   S: a statement
   G: an SDG
   WorkList: set of vertices
   v: a vertex
begin
   G := BuildSDG(P)
   do
      G := Slice(G, VertexFor(S))

      WorkList := { Vertices that a rewriting
                    rule applies to }
     while WorkList ≠ ∅ do
        Select and remove a vertex v from WorkList
        Apply the rewriting rule for v,
            updating G and WorkList
     od
   while no changes
   G := MakeFeasible(G)
   SlicedP := Reconstitute(G)
   return(SlicedP)
end
```

Figure 2. Function AmorphousSlice computes an amorphous slice of program P taken with respect to statement S.

Table 1: Example Rewriting Rule

Rule	Explanation
flow edge removal	For a flow-dependence edge from an assignment vertex v labeled "$a = e$" to a vertex u, the expression e can be propagated along the edge to u and the edge subsequently removed. For this rewriting to be allowed, u must be the target of exactly one flow-dependence edge for variable a. The SDG is updated by copying the incoming flow-dependence edges of v to u and rewriting the label of u by substituting e for a. If u is an assignment vertex that assigns to a, then its rewriting requires rewriting all flow-edges from v in order to avoid situations in which the graph cannot be transformed back into a sequential program. For example, consider removing the flow edge from the second to the fourth assignment in the sequence "$i = 1;\ i = i + 1;\ a = i;\ i = i + 1;$."

a slice is computed with respect to a variable, not a concept such as "array accesses safety." Therefore, the first step in computing a safety slice, is to augment the program to make array safety explicit [7, 10]. The augmentation includes assignments to a new variable *safe* such that *safe* has value **true** iff the program has no out of bounds array references.

In more detail the augmentation first adds, to the beginning of the program, the assignment "*safe* = **true**" (at the start of execution there have been no array bounds errors). At the end of the program the statement "*print safe*" is added. Finally, immediately before each array reference, $a[i]$, the assignment statement

$safe = safe\ \&\&\ i \geq 0\ \&\&\ i < upper\_bound(a)$

is added (this assumes that arrays start at index 0).

In the ideal case, the amorphous slice of the augmented program, hereafter referred to as the *safety slice*, is one of the following:

print **true**
// the original program has no array bounds errors

or

print **false**
// the original program has an array bounds error

In the more common case, the safety slice produces a program of more than one line. This program computes the safety of the original program. That is, for a given input, this program computes whether or not the original program contains an array bounds error.

The remainder of this section summarizes the computation of a safety slice. Figure 3 shows a program after it has been augmented. A series of slices and graph rewritings remove all of the code associated with the first the third loops [3, 9]. Consider the rewriting of the flow edges in the middle loop. The initial loop, shown in Figure 4(a), has three flow edges that are candidates for flow-edge removal. Figure 4(b) shows the result of removing the edge labeled $flow^1$. After this rewriting, the flow-dependence edge from the first assignment of *safe* to the second has been copied to the final assignment (the copy is labeled $flow^2$ in Figure 4(b)). This rewriting also updates the expression labeling the target of edge $flow^1$ to be "$safe = safe\ \&\&\ x \geq 0\ \&\&\ x \leq 127\ \&\&\ x \geq 0\ \&\&\ x \leq 127$" which simplifies to "$safe = safe\ \&\&\ x \geq 0\ \&\&\ x \leq 127$" as shown in Figure 4(b). Figure 4(c) shows the result of removing the edges labeled $flow^2$ and $flow^3$. In all three removals, the source vertex has no other outgoing edges and thus is removed from the graph as part of the rewriting.

The remaining while-loop cannot be further rewritten because the value of *source* is not known until runtime. The final safety slice is

```
scanf("%s", source);
safe = 1;
i = 0;
while(source[i] != '\0')
{
   safe = safe && i>=0 && i<=MAX &&
            source[i]>=0 && source[i]<=127;
   i = i+1;
}
printf("safe = %d", safe);
```

The location of this code immediately tells an engineer that the array references in the first and third loops are safe. The code itself indicates that the middle loop may cause array subscript errors.

In particular, the program outputs **false** if the following condition is ever violated:

```
i>=0 && i<=MAX &&
source[i]>=0 && source[i]<=127
```

```
/* Safety Slice Testing Program */
safe = 1; /* starts out safe */
[printf("Enter a string");]
scanf("%s",source);
i = 0;
while (i<128)
{
   safe = safe && i>=0 && i<=127;
   result[i] = 0;
   i=i+1;
}
i = 0;
safe = safe && i>=0 && i<=MAX;
while (source[i] != '0')
{
   safe = safe && i>=0 && i<=MAX;
   x = source[i];
   i = i+1;
   safe = safe && x>=0 && x<=127;
   safe = safe && x>=0 && x<=127;
   result[x] = result[x]+1;
}
[b = 0;]
bc = 0;
i = 'a';
while (i<128)
{
   safe = safe && i>=0 && i<=127;
   if (result[i] > bc)
   {
      [b = i;]
      safe = safe && i>=0 && i<=127;
      bc = result[i];
   }
   i = i+1;
}
[printf("Most frequent: %c ",b);]
[printf("occurs %d times\n",bc);]
printf("safe = %d",safe);
```

Figure 3. An augmented program that computes the most frequently occurring ASCII character in its input. The slice of this program with respect to printing of *safe* **excludes only the boxed statements. Note how statement deletion alone removes only a small portion of the program. (The constant MAX is the declared size of the string** *source***.)**

The present implementation does not recognize that i will always be greater than or equal to 0 (although there is no technical reason why it could not). The remaining conditions highlight two invalid assumptions made by the original programmer: first, the string read into *source* is assumed to be no longer than *MAX* characters, and second, the input characters are assumed to be ASCII characters within the range $0 \cdots 127$.

The SDG construction and slicing phases of the present implementation are capable of handling essentially all of ANSI C. This is done using the code analysis tool CodeSurfer [6]. The rewriting is less complete. It presently handles only a small number of C language features. It is capable of producing the examples used in the experiment and in this paper.

4. The Experiment

An experiment was designed and administered to assess the effectiveness of amorphous slicing in assisting an engineer identify properties of interest in a program. The experiment uses both qualitative and quantitative methods to gauge the effectiveness of safety slicing in helping identify array subscript violations.

The section first discusses the experimental setup and design. It then considers numerical results from the three classes of subjects who took part in the experiment. Finally, selected answers to subjective questions are presented to help provide a context for the numerical data.

4.1. Experimental Setup and Design

The experiment is designed to test the general hypothesis that a safety slice assist in program comprehension. Specifically, two hypotheses are under test in this experiment:

(1) Safety slices make human debugging of array safety more accurate, and

(2) Safety slices make human debugging of array safety more efficient.

Seventy nine subjects in six groups took part in the study. The groups included students half way through the first, second, and third years of study at Loyola College. Students from each year were divided into a control group and an experimental group.

For the first-year students, the two groups were created by simply using two separate sections of the same course (taught by the same instructor). It was believed that this would provide sufficient random distribution of talent. Unfortunately this assumption was incorrect. This was obvious from the student's demeanor and confirmed by the instructor. The second and third-year students were

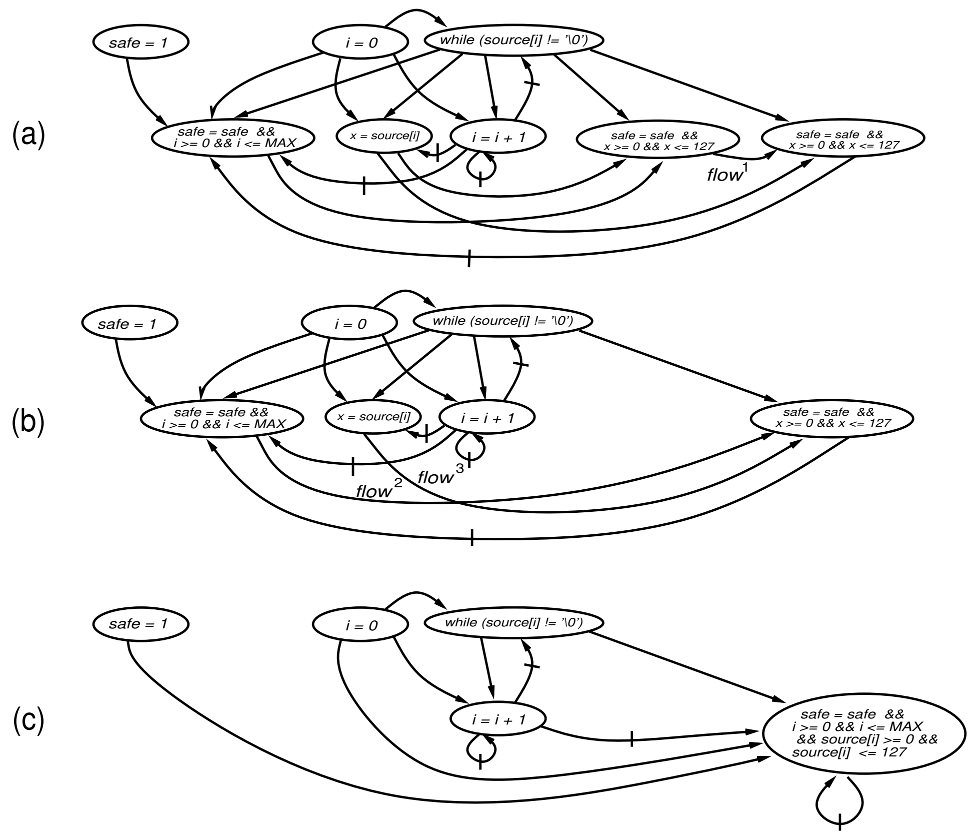

Figure 4. Part of the SDG for the middle while loop of the example program. The vertex labeled "$safe = 1$" is not from the original program. It is the result of rewriting the first loop. The vertex for the statement "$result[x] = result[x] + 1$" is not shown as it was not in the second traditional slice.

divided by class standing; thus, providing more uniform distribution of talent.

To test the two hypothesizes, all groups were given the definition of an array bounds violation, a worked example, and a brief 15 minute lecture on array bounds violations. The same person gave this lecture to all six groups. For the first-year students, this was in back to back classes. For the second and third-year students it was given simultaneously to both groups. In all cases, the same set of notes was followed to avoid biasing any particular group. In order to train the subjects in the use of safety slices, the lecture included an explanation of safety slicing and how to use the safety slice in searching for array violations.

All groups were then given seven programs containing array subscript violations. In addition, the experimental groups received the safety slice for each of the seven programs. The first five programs were training programs designed to determine how well the subjects understood their task (the third of these is described below). The sixth program, shown in Figure 3, was used to generate the results presented in this section. The last program was a "difficult" problem designed to test the likelihood subjects would admit when they were guessing. Following the seven programs was a questionnaire, which asked participants subjectively to rate their performance, confidence, and the difficulty of the problems. Each group was then given the remainder of the lecture period to find array subscript violations.

Figure 5 shows the third training program and its answer sheet. In this program, the second reference is a classic out of bounds array reference. The safety slice received by the experimental group is included in the right-hand column. The first assignment to *safe* is place

PROGRAM P3 (Training Programs 3)

```
                  /* Original Program */            /* Array Safety Slice */

                  #define MAX 10                    #define MAX 10
                  main()                            main()
                  {                                 {
                     int a[MAX];                       int i;
                     int i;
                     int looking_for;                  int safe = 1;   // start out safe

                     // read in array
                     // read in looking_for

                     for(i=0; i<MAX; i++)
REF 1                   if (a[i] == looking_for)       safe = safe;
                           break;

REF 2                if (a[i] == looking_for)          safe = safe && MAX < MAX;
                        printf("found");
                     else
                        printf("not found");
                                                       printf("safe = %d\n", safe);
                  }                                 }
```

Answers Sheet for Program 3

(1)
- ○ safe
- ○ can be too small
- ○ can be too big
- ○ can be too small and too big

confidence 4=high ... 0=low=guess

4	3	2	1	0
□	□	□	□	□

reason: ____

(2)
- ○ safe
- ○ can be too small
- ○ can be too big
- ○ can be too small and too big

confidence 4=high ... 0=low=guess

4	3	2	1	0
□	□	□	□	□

reason: ____

Figure 5. The program and answer sheet for Training Program 3.

holder. The second assignment was used to check if subjects in the experimental group understood what the safety slice represented well enough to realize the implication of *MAX < MAX* being **false**. The current implementation actually removes the first assignment and replaces the second with "*safe* = **false**."

4.2. Overall Results

Overall, there is statistical evidence that safety slicing assists in program comprehension. For the most part, it improved subject's ability to identify array access violations. While subjectively, safety slicing also improved efficiency, the statistics to not back this up.

The data related to Hypothesis 1 for all six groups is summarized in the following table. The row labeled "First-Year Experimental Y" represent a subset of the first-year experiential group who answered "yes" they understood what a safety slice represented. With the exception of the first-year experimental group, mean scores increase as the experience level of the subjects increases and with access to the safety slice.

Results Summary

Group	$\bar{x}$	s	n
First-Year Control	4.64	1.92	22
First-Year Experimental	4.39	2.02	18
First-Year Experimental Y	5.00	1.85	8
Second-Year Control	5.27	1.35	11
Second-Year Experimental	6.33	0.87	9
Third-Year Control	6.00	0.89	11
Third-Year Experimental	6.50	0.93	8

The graph in Figure 6 shows the mean and 95% confidence interval for each group. In the graph, the short vertical bar represents the mean and the longer horizontal bar represents the confidence interval. The figure should not be used for direct comparison of groups; this requires a test of the difference of two means. However, it does show with 95% confidence where the mean for each groups is expected to lie. Furthermore, the trend in the length of the lines illustrates how the standard deviation is decreasing with experience.

The data related to Hypothesis 2 are only meaningful for the second-year students. For different reasons the first and third-year students didn't produce meaningful timing numbers. These reasons are discussed below. Subjectively, the experimental groups finished faster or completed more of the problems in the time alloted.

The remainder of this section compares various groups from the study. First combined and inter-year results are considered, then each year is considered independently. In each case the statistical test used is a one-sided t-test involving the difference of two means (t-tests are used because some of the groups are small). The null hypothesis in each case is $\mu_1 - \mu_2 = 0$ and the alternative hypothesis is $\mu_1 - \mu_2 < 0$ (*i.e.*, the experimental group has a higher mean score), except for the timing question where the alternative hypothesis is $\mu_1 - \mu_2 > 0$ (*i.e.*, the experimental group is faster). In each case the p-value, the common standard deviation, and the degrees of freedom (*d.f.*) are given. The p-value can be interpreted as follows: values less than 0.01 are highly significant, values less than 0.05 are significant, and values less than 0.10 are weakly significant.

The analysis assumes that the parent populations have normal deviations; t-tests are know to be robust against non-normality of the data. The analysis also assumes that the standard deviations σ_1 and σ_2 for the two populations are unequal (hetroscedastic). Repeating the analysis assuming they are equal (homoscedastic) does not affect any of the conclusions. Numerically the p-values change (some up and some down) by no more than 0.003.

Taking all subjects together, safety slicing does not significant improve the identifying of array bounds violations ($p = 0.28$, $s = 1.76$, *d.f.* $= 77$). Replacing the first-year students with the subset of first-year students who said they understood what safety slicing was, produces a dramatic turnaround. Here safety slicing clearly helps ($p = 0.016$, $s = 1.57$, *d.f.* $= 67$). Considering only the more advanced second and third year students safety slicing is quite effective ($p = 0.008$, $s = 1.04$, *d.f.* $= 35$).

A few other intergroup comparisons are interesting. First, the third-year experiential group does not do significantly better than the second year experiential group ($p = 0.35$, $s = 0.89$, *d.f.* $= 15$). However the third-year control group does do somewhat better than the second-year control group ($p = 0.077$, $s = 1.14$, *d.f.* $= 20$). Taken together, these two may indicate that safely slicing was able to take the second-year students to the level of the third year students. Finally, some not surprising results: The third-year experiential group clearly does better than the second-year control group ($p = 0.016$, $s = 1.19$, *d.f.* $= 17$) and the second-year experiential group does better than the first-year control group ($p = 0.001$, $s = 1.70$, *d.f.* $= 29$), the first-year experiential group ($p = 0.001$, $s = 1.77$, *d.f.* $= 25$), and the first year experiential "Y" group ($p = 0.046$, $s = 1.41$, *d.f.* $= 15$).

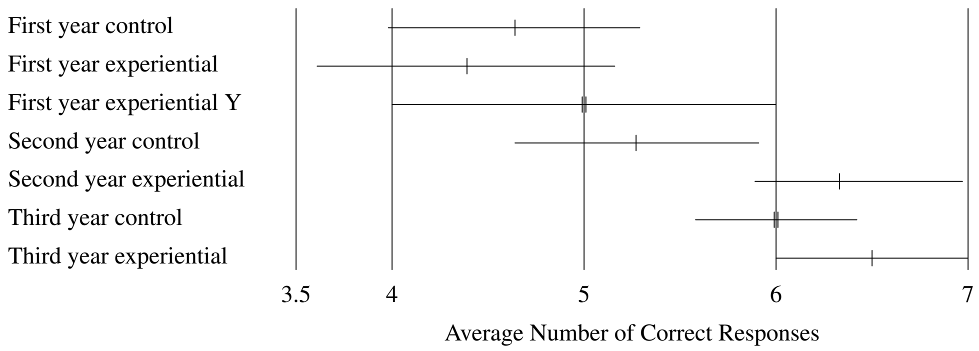

Figure 6. The mean and 95% confidence intervals for the six groups.

4.3. First-Year Students

Each class is now considered independently. The following table summarizes results from the first-year students. The column labeled “score” summarizes the number of correctly identified array references from the program shown in Figure 3. The column labeled “P3” is the third training program shown in Figure 5. Note that not all students completed all problems, which accounts for the different values of n in the columns labeled “P3” and “score.”

First-Year Students

	Experimental		Experimental Y		Control	
	P3	score	P3	score	P3	score
n	23	18	12	8	24	22
% correct	87%	63%	90%	71%	82%	66%
mean score	6.12	4.39	6.33	5.00	5.71	4.64
stdev	0.56	2.02	0.89	1.85	0.44	1.92

The mean for the control group is higher than the experiential group, but not significantly so ($p = 0.65$, $s = 1.98$, $d.f. = 38$). On training problem P3, the experiential group did slightly better than the control group ($p = 0.10$, $s = 0.50$, $d.f. = 46$). This provides some evidence that although they may not have been ready to use the safety slice with more difficult problems, it did provide some assistance on a problem closer to their level.

The timing data related to Hypothesis 2 was impossible to accurately collect because none of the control group were finished when, five minutes before the end of the period, they were asked to goto the questionnaire. About one third of the experimental group finished before the end of the period. The rest were asked to goto the questionnaire before finishing.

Subjectively, the first-year students were less able to stay on task than the more advanced students. After approximately 20 minutes some were noticeably fidgety. One potential reason for this is their inexperience with arrays. For example, given the code

```
int a[10], b[10];
...
for(i = 0; i < 10; i++)
  a[i] = a[i] * b[i];
```

The left-hand side reference to $a[i]$ was marked as unsafe because “$a[i]$ *(not i)* might be larger than 10.” Other similar responses were given.

4.4. Second-Year Students

The strongest results within an individual year were obtained with the second-year students. The following table details their scores and time taken. In the time column, an entry of “?” represents either no value was recorded for the finishing time, or in one case, a subject entered an impossibly low number.

Second-Year Students

	Experimental		Control	
	score	time	score	time
	5	20	3	16
	7	18	3	25
	7	15	5	30
	5	23	6	15
	6	?	7	?
	6	20	5	30
	7	?	6	?
	7	21	6	17
	7	15	5	?
			5	18
			7	25
% correct	90%		75%	
mean	6.33	18.86	5.27	22.00
stdev	0.82	2.80	1.29	5.83

The experiential group outperformed the control group ($p = 0.024$, $s = 1.16$, $d.f. = 18$). While they also had a faster finishing time, it is not significantly faster ($p = 0.18$, $s = 5.07$, $d.f. = 14$).

When asked if the safety slice was helpful seven of the nine subjects from the experimental group answered “yes.” One answered “sometimes” and one left the answer blank. Unlike the first-year students, considering only those subjects who said “yes” makes no difference to the results.

4.5. Third-Year Students

The following table summarizes the responses of the third-year students. As explained below, this incarnation of the experiment was given in less than ideal circumstances.

Third-Year Students

	Experimental	Control
	score	score
n	8	11
% correct	93%	86%
mean	6.50	6.00
stdev	0.87	0.85

Although their mean is higher, the experiential group does not outperformed the control group ($p = 0.13$, $s = 0.91$, $d.f. = 17$). In the experimental group, half the wrong answers were given correctly then scratched out and replaced with the wrong answers. No reason was given for either decision.

Each subject in this group was asked to rank their confidence in their answers from 0 (I guessed) to 4 (I'm sure). While not all subjects filled in the confidence ranking, of those who did from the experimental group, only one subject marked anything other than a 4 for an array reference that was identified as safe by the safety slice. This subject marked a 0 (a guess) on two such references and got them wrong. The experimental group's other confidence rankings—for array references where the safety slice provided only guidance—were mostly 3's with some 2's. In contrast, the confidence rankings of the control group were slightly lower and more evenly distributed over all possible responses. One thing this indicates is that the control group is working where they need not.

There are two reasons for caution in interpreting the results for third-year students. First, this group was rushed, so the timing information is not meaningful and is not reported (subjects were told to stop and go to the end to fill out the questionnaire). Subjectively, the experiential group was farther along when time ran out.

The second reason for caution was brought to light in response to the question "Overall how confident are you of your answers?" It turns out that the answer "not completely (no sleep)" was not uncommon as the experiment was administered the same day as a large software engineering project was due.

4.6. Subjective Questions

The questionnaire at the end of the experiment contained qualitative questions. Responses to these questions provide a subjective context for the numerical data. In general, the responses indicated that most subjects who used the safety slice technique liked it.

Responses to the question "how confident are you in your answers?" indicated that the safety slice, when understood, was helpful. Overall the experimental groups seemed to be more confident, but the difference was not overwhelming.

In response to the question "did you find the safety slice helpful in finding errors?" thirteen first-year students answered "yes," eight answered "no," and the rest said things ranging from "somewhat" to "sometimes." With the second-year students seven answered "yes," one "sometimes," and one gave no answer. Finally, all of the third-year students who answered responded "yes."

Individual responses to this question included "yes and no, it helped me find some, but it confused me too" (this answer, by a first-year student, points to the need to better training). Another first-year student responded "[the safety slice was] helpful, especially in the harder programs." A second-year student responded "somewhat, it gave me an idea as to where the errors might have been," (which is exactly what a safety slice is designed to do). A third-year student responded "it is nice to see the possible problems with your code displayed."

5. Summary and Future Work

The utility and power of program slicing comes from its ability to assist in tedious and error prone tasks such as program debugging, testing, parallelization, integration, software safety, software understanding, software maintenance, software metrics, and program comprehension [2, 13]. Slicing does this by extracting an algorithm—whose computation may be scattered throughout a program—from intervening irrelevant statements.

Amorphous slicing is useful in reducing the size of a slice when the result need not be composed solely of statements from the original program. Section 3 describes how an amorphous slice of a program augmented to track array references is used to produce the *safety slice*.

An important aspect of this technique is that it works with, and produces as answers, programs. By using a programming language, which is a language an engineer should be comfortable with, the technique avoids the need for new, potentially ill-understood, formalisms.

Furthermore, recasting the problem as one of simplification, circumvents the undecidability problem because the simplification process consists of approximating the ideal simplest program. This allows the technique to make progress in deciding the truth or otherwise of a proposition, in cases where it is impossible to fully decide. Safety slices provide a way of partially automating support for comprehension, based on the formulation and investigation of hypotheses.

The experiment detailed in Section 4 demonstrates that the safety slice can aid a software engineer in finding array bounds violations. For all but the first-year students, the safety slice improves the accuracy for the subjects in finding array subscript violations. Furthermore, the control groups seemed to take longer and be less sure of their answers. Future experiments will investigate the techniques external validity (*i..e*, does safety slicing aid students elsewhere or practicing software engineers).

Acknowledgments
GrammaTech (www.grammatech.com) provided *CodeSurfer*^{tm} upon which the implementation is based. The reviewers, especially Reviewer 3 clarified the presentation. Erin Binkley proof read the paper.

6. References

1. Binkley, D., "Precise executable interprocedural slices," *ACM Letters on Programming Languages and Systems* **2**(1-4) pp. 31-45 (1993).
2. Binkley, D. and Gallagher, K., "Program slicing," *Annals of Computing*, (1996).
3. Binkley, D., "Computing amorphous program slices using dependence graphs," *Proceedings of the 1999 ACM Symposium on Applied Computing,* (San Antonio, Texas, February 1999)., pp. 519-525 (February 1999).
4. Choi, J. and Ferrante, J., "Static slicing in the presence of goto statements," *ACM Transacton on Programming Languages and Systems* **16**(4) pp. 1097-1113 (July 1991).
5. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* **9**(3) pp. 319-349 (July 1987).
6. Grammatech Inc., *http://www.grammatech.com/products/codesurfer/codesurfer.html*. 1999.
7. Harman, M. and Danicic, S., "Using program slicing to simplify testing," *Journal of Software Testing, Verification, and Reliability* **5** pp. 143-162 John Wiley & Sons Ltd, (1995).
8. Harman, M., Sivagurunathan, Y., and Danicic, S., "Analysis of dynamic memory access using amorphous slicing," in *to appear in IEEE International Conference on Software Maintenance,* (Washington, DC, Nov, 1998), IEEE Computer Society, Washington, DC (1998).
9. Harman, M., Binkley, D., and Danicic, S., "Amorphous programs slicing," *Loyola College Technical Report CS-TR-00-23,* submited for publication, (1999).
10. Harman, M., Fox, C., Hierons, R., Binkley, D., and Danicic, S., "Program simplification as a means of approximating undecidable propositions," pp. 208-217 in 7th *IEEE International Workshop on Program Comprehension,* (Pittsburg, PA, May, 1999), IEEE Computer Society, Washington, DC (1999).
11. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).
12. 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).
13. Tip, F., "A survey of program slicing techniques," *Journal of Programming Languages* **3**(3) pp. 121-189 (September 1995).
14. Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).