An Implementation of and Experiment with Semantic Differencing

David Binkley Rob Capellini L. Ross Raszewski Christopher Smith
Computer Science Department
Loyola College in Maryland
4501 N. Charles Street
Baltimore, MD 21210 USA
+ 1 410 617 2881
{binkley,rcapelli,lraszews,csmith}@cs.loyola.edu

Abstract

Software maintainers face a wide range of difficult tasks including impact analysis and regression testing. Understanding semantic relationships, such as the semantic cohesiveness in a program or the semantic differences between two programs, can help a maintainer address these problems. However, semantic analysis is a difficult problem. For example, few semantic differencing algorithms and even fewer implementations exist.

The first semantic differencing implementation for the C language is presented and studied. A large collection of semantic differences of 10 program are computed. The average size reduction was 37.70%. This study illustrates the practicality of semantics differencing. Finally, the application of semantic differencing in the area of program testing and impact analysis is considered.

1. Introduction

Software maintainers face difficult tasks such as regression testing: retesting of a program after an enhancement or a bug fix, and impact analysis: determining the complexity and the affect of a proposed change to a piece of software. Semantic differencing can be applied to these problems.

Computing semantic differences is a difficult task. Few algorithms exist [15, 5]. To date no implementation for a full featured programming language has been presented. This paper describes an implementation for the C language based on two tools: Grammatech's CodeSurfer [11] and Cdiff [37]. CodeSurfer is a program analysis tool designed to perform deep semantic analyses. Among these analyses is program slicing [35, 33]. Cdiff is a tool for identifying syntactic differences between C programs.

In simple terms, the implementation takes as input two programs `old` and `new`, and outputs a third program `differences`. The significance of `differences` is that it captures the semantic (rather then syntactic) differences between `old` and `new`.

The implementation is used in a study of a collection of programs. At a low level the study investigates the time taken to compute `differences` and, more importantly, the size of `differences` relative to the size of `new`. At a high level, the study seeks to illustrate the usefulness of semantic differencing.

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` [7]. The capturing of semantics 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 be violated when, for example, the location at which a program is loaded affects its behavior [7].

The remainder of this paper is organized as follows: Section 2 presents some background material on the algorithm for computing `differences`. The main technical contributions of this paper appear in Sections 3 and 4, which describe the implementation of the algorithm, and the results of the experiments. This is followed in Section 5 by a brief look at two uses for semantic differencing. Finally, Section 6 discusses related work, and Section 7 summarizes the paper and discusses future work.

This research is supported by National Science Foundation grants CCR-9411861 and CCR-9803665.

2. Background

This section provides background on the algorithm used to compute `differences` [7]. It first describes the system dependence graph and then the interprocedural slicing operators [17] used by the algorithm. It concludes with the algorithm for computing `differences`.

The system dependence graph (SDG) is a collection of procedure dependence graphs (PDGs) connected by interprocedural control- and flow-dependence edges [17, 6]. Each PDG contains vertices, which represent the components (statements and predicates) of a procedure, and edges, which represent the control and flow dependences between components. The dependences in a PDG are safe approximations to the semantic dependences found in the program [24], which are generally not computable and therefore must be (safely) approximated.

Backward and forward interprocedural slices [35, 17] are used to compute `differences`. For a slice on SDG vertex v, an interprocedural backwards slice contains those vertices that potentially affect a computation represented at v. It can be computed using two passes [17]: First, a *b1* slice, identifies all vertices of the slice that are not in called procedures. Second, a *b2* slice, identifies the necessary vertices from called procedures. An interprocedural forward slice, which contains those vertices that are potentially affected by a computation, is computed in two passes, named *f1* and *f2*, that mirror those of backwards slicing.

Program `differences` is computed by first building the SDGs for `old` and `new`. The algorithm then gathers a series of slices of `new`'s SDG starting with a set of points called the *directly affected points*. The final step of the algorithm is to project out of `new` those statements represented by vertices in these slices [7]. The complete algorithm for computing `differences` is stated in Figure 1. It computes the following quantities:

Directly Affected Points
: The set of *directly affected points* is a safe approximation to the set of components from `new` directly influenced by the change to `old` [6].

Affected Points
: The set of *affected points* is a safe approximation to the vertices of `new` that represent components whose behavior (*interpretations* [24]) may differ in `new` and `old`.

Strongly Affected Points
: The set of *strongly affected points* is a safe approximation to the components of `new` that are potentially affected regardless of the calling context.

Δ
: The set Δ is a safe approximation to the set of components necessary to capture the behavior of the affected points [6].

---

function ComputeDiffereces(`old`, `new`) returns a Program
declare
  `old`, `new`, `differences`:Programs
  $G_{\texttt{old}}, G_{\texttt{new}}$: SDGs
  DirectlyAffectedPoints, *StronglyAffectedPoints*,
    AffectedPoints, Δ : sets of SDG vertices
begin
  $G_{\texttt{old}} := \mathit{BuildSDG}(\texttt{old})$
  $G_{\texttt{new}} := \mathit{BuildSDG}(\texttt{new})$
  $\mathit{DirectlyAffectedPoints} := \mathit{FindDAP}(G_{\texttt{old}}, G_{\texttt{new}})$
  $\mathit{StronglyAffectedPoints} := \mathit{f1}(G_{\texttt{new}}, \mathit{DirectlyAffectedPoints})$
  $\mathit{AffectedPoints} := \mathit{f2}(G_{\texttt{new}}, \mathit{StronglyAffectedPoints})$
  $\Delta := \mathit{b2}(G_{\texttt{new}}, \mathit{AffectedPoints}) \cup$
    $\mathit{b2}(G_{\texttt{new}}, \mathit{b1}(G_{\texttt{new}}, \mathit{StronglyAffectedPoints}))$
  $\texttt{differences} := \mathit{ProjectProgram}(\texttt{new}, \Delta)$
  return `differences`
end

Figure 1. The algorithm for computing `differences`.

`differences`
: `Differences` is an executable program that captures the computation of `new` that are different from those of `old`.

Example. The program, shown in Figure 2, 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 the directly affected points. These are the vertices that make up procedure *Add*, whose behavior is different (completely new) when called from *Multiply*. (In this simple example, the set of strongly affected points is the same as the set of directly affected points. If the change had affected an existing computation in, for example, `main`, then the vertices representing this computation would be strongly, but not directly affected points.) The graph Δ includes those vertices in various backwards slices taken with respect to the affected (and strongly affected points). In the example, this includes the computation of i. Finally, the resulting program `differences` is produced by projecting out of `new` those statements represented in Δ. The result is highlighted in Figure 2. One thing illustrated by this example is the need for non-affected points in `differences`. Here, the computation of `i` is not affected by the change, but it is required to capture the computation of `prod` in an executable program.

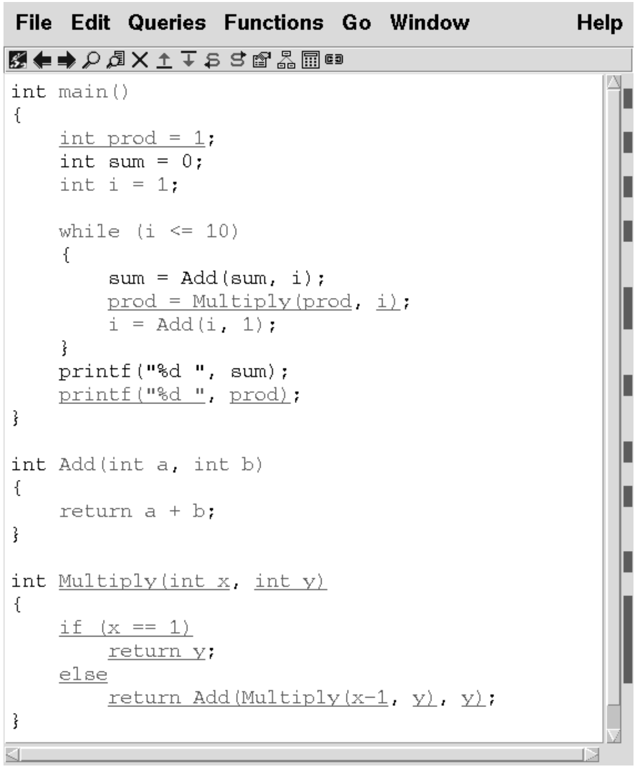

Figure 2. The product program. `New` was created by enhancing `old` with the computation of `prod`. The statements that make up `differences` are highlighted in red by CodeSurfer and appear light gray in the figure. Statements representing directly affected points are underlined. (Function `Multiply` assumes its arguments are positive.)

3. Implementation

This section describes a prototype implementation of the algorithm for computing `differences`. The implementation glues together two existing tools, CodeSurfer [11] and Cdiff [37], using bits of Scheme and C code. Most of the work is done by CodeSurfer, a code analysis tool that, among other things, allows a software engineer to take and combine various program slices. CodeSurfer represents programs as SDGs and provides access to these SDGs through a scheme API.

The implementation has one significant deviation from the algorithm. The directly affected points are computed from `old` and `new` and not their SDGs. This is done, in the first step of the implementation, by Cdiff, which uses the grammar of the C language and dynamic programming to identify components of two C programs that are different. For the programs used in the study it was determined by inspection that Cdiff precisely identified the directly affected points. In theory Cdiff produces a superset of the directly affected points, but as Cdiff computes this set quickly and the over approximation has so far been small or non-existent, it makes a good choice for the implementation.

A small C program transforms the output from Cdiff into scheme code denoting the character positions in `new` for the directly affected points. This scheme code is used by CodeSurfer to identify the directly affected points.

The next step of the implementation is to build the SDG for `new`. This is done by CodeSurfer. The construction cost for building `new`'s SDG is polynomial in the size of `new` [17, 6]. However, this step can be performed "off-line" if the SDGs for programs are maintained as part of a programming environment [23].

Using the character position information obtained from Cdiff, the vertices in the SDG for `new` that are directly affected points are identified. The set of vertices Δ is then constructed using the function delta written in CodeSurfer's Scheme API. This computation is linear in the size of Δ. The resulting set is output as a collection of character positions in the source of `new`. These locations represent the components of `new` that will be in `differences`.

Finally, a small C program extracts `differences` from `new` using these locations. This C program also performs some syntactic cleanup as CodeSurfer does not output its slices as syntactically complete C programs. In the present implementation, this cleanup is incomplete; thus, some minor hand editing is required to produce the final program `differences`. It is important to note that this editing is not fundamental to the algorithm or the implementation. It is presently required because CodeSurfer does not maintain complete syntax with each node. A future release of CodeSurfer that includes sufficient abstract syntax information will allow this process to be completely automated. Presently, this editing entails adding function return types, omitting the keyword else when the false-block is empty, and inserting or moving braces ('{' and '}').

The utility of CodeSurfer and to a lesser extent Cdiff is illustrated by the size of the scheme and C code required to construct the system. A little under 200 lines of scheme code is all that is required to implement the computation of Δ. The two small C programs are each under two pages. The scheme API was quite useful in developing and testing the prototype.

4. The Study

This section presents an imperial study of the implementation described in Section 3. The time taken to compute Δ and its size for two groups of programs are discussed in Sections 4.1 and 4.2. The study also considers statistics related to the size of the system dependence graph relative to the size of the source program. As explained below, `differences` is only produced for the first group.

4.1. Group 1: Small Programs

Each program in the first group includes multiple versions each with multiple test cases. This group has been used before in regression testing studies [30]. Therefore, regression testing is used as a motivating example application in this section.

The programs, provided by Siemens Corporate Research, have between 7 and 41 different versions and range from 145 to 514 non-blank lines of code. Each is composed of 7 to 21 functions and was designed to contain realistic faults that were neither too hard nor too easy to detect [28].

Consecutive versions of each program were used as `old` and `new` and input into the implementation. Table A reports the average, best case, and worst case reduction for each program. In the table, the column labeled "versions" is the number of versions of the program and the column labeled "size" is the average number of lines in `new` over all the versions of the program. For these small programs, SDG build time ranged from 0.24 to 0.80 seconds. This corresponds to the translation of 483 to 905 statements per second. The computation of Δ took no measurable time. The entire computation of `differences` (not including the hand editing) took less than one second for each run.

For small "single-thought" programs `differences` is expected to contain most of `new`. However, the table contains some encouraging numbers. As shown, the size reduction ranges from 9 to 95 percent with the average over all programs being 26 percent. Preliminary results from a related study designed to determine if computing and then testing `differences` is more efficient than testing `new` indicates that using differences can significantly reduce the cost of regression testing.

Table A

program	versions	size (loc)	percent size reduction worst	average	best
calc	11	166	9%	15%	45%
print_t1	7	513	38	39	39
print_t2	10	468	33	34	38
replace	32	514	11	12	19
schedule1	9	362	27	27	28
schedule2	9	280	19	33	95
tcas	41	145	17	24	27
average	17	350	22	26	42

4.2. Group 2: Larger Programs

The second group consists of 10 larger programs (ranging in size from 3,000 to 22,000 lines of code). It is easier and less ambiguous to count SDG vertices in Δ than "statements" in `differences`; thus, the size reduction results in this section are given in terms of Δ's size.

The scheme implementation, which was useful for prototyping the algorithm, proved too slow for these programs; thus, it was recoded in C. The resulting system can efficiently handle programs up to about 50,000 lines of code. Note that a recent release of CodeSurfer exposed the four slicing operators through the scheme API; thus, obviating the need for the C rewrite.

Multiple variants of the programs studied in this section were not available, so for each program, Δ was computed for each vertex in the program's SDG. Only vertices that represented program source code were considered. These vertices correspond to parts of the program that could be discovered as directly affected points by Cdiff. They omit vertices that represent code from libraries, such as libc, and vertices that represent passing global variables to functions, which are modeled as additional procedure parameters in the SDG.

For some programs, the vertices that represent globals accounted for a significant portion of the vertices in the SDG. For example, flex has 54,788 vertices, of which 40,742 represent global variables.

To test the impact of not computing Δ for these vertices, the results from the remaining 10,867 selected vertices were compared with those produced using all 54,788 vertices (at a little under 3/4 of a second for each computation; it takes just under 11 hours to complete the analysis for all of flex). The results of the two are essentially identical. Using all vertices produces a stretched version of the selected vertices results. That as, if the x-axis was scaled to account to the different number of points, the graphs would be the same. Thus, focusing only on vertices that represent "real" code is presumed to not affect the results presented in this paper.

The remainder of this section discusses three tables and a collection of graphs that present results for the 10 programs. The first table, Table B, presents statistics on SDG size. It includes for each program, the size (in lines of code), the SDG size (in vertices), and the number of vertices per line of code. Averaged over all 10 programs 6.28 SDG vertices are used to represent each statement.

Table C details the computation time for the various parts of the `differences` algorithm. The column labeled "pace" is the number of lines of code processed per second by CodeSurfer on a 450MHz Pentium II using 256Mb of Memory. On average this is about 250 lines per second, or 25 to 50 percent of the speed of gcc with full optimization. Given the 12-fold improvement in CodeSurfer's speed from Version 1.0 to Version 1.1, it is reasonable to expect that future tuning will make its speed comparable to that of highly optimizing compilers.

The most important information in Table C is the time taken to compute Δ. For each computation of Δ, the user, system and elapsed time were recorded. Only the user time is shown in Table C. The recorded system time is zero in 96.5% of the runs. In the remaining cases, it gets as high as about 20% of elapsed time, and is most likely related to unfortunate paging behavior. Elapsed time mirrored user time, except when some daemon woke up and caused the elapsed time to spike up. The rationale for selecting user time is evident after examining the graphs of the three times for each program. In the remainder of the discussion, compute time refers to user time.

Table B			
program	LOC	SDG size	vertices / LOC
a2s	22,068	57,922	2.62
acct	10,182	110,860	10.89
barcode	5,848	19,562	3.35
bc	8,364	70,502	8.43
copia	3,488	46,130	13.23
ed	13,578	98,479	7.25
flex	21,543	53,418	2.48
oracolo2	14,382	92,200	6.41
prepro	14,330	95,757	6.68
termutils	7,006	10,161	1.45
average	12,079	65,499	6.28

Table C						
		SDG build		Δ compute time		
		time	pace	min	ave	max
program	LOC	(sec)	(LOC/sec)	(sec)	(sec)	(sec)
a2s	22,068	106	208	0.05	0.94	2.26
acct	10,182	99	103	0.00	0.89	5.07
barcode	5,848	17	344	0.02	0.16	0.26
bc	8,364	64	131	0.06	2.34	3.59
copia	3,488	9	388	0.04	0.72	1.87
ed	13,578	243	56	0.02	4.29	7.76
flex	21,543	120	180	0.05	0.72	2.24
oracolo2	14,382	44	327	0.07	1.17	3.54
prepro	14,330	44	326	0.08	1.22	3.60
termutils	7,006	15	467	0.00	0.05	0.11
average	12,079	76	253	0.04	1.25	3.00

The last three columns in Table C give the time to compute Δ. The maximum time occurs when the entire graph must be walked to obtain Δ. The minimum time occurs typically for dead code where Δ contains only a handful of vertices. Finally, the average time is the average over all runs of the program. Averaged over all the programs, the speed of the Δ computation is an impressive $47,270$ vertices per second.

In addition to the computation of Δ, computing `differences` requires running Cdiff. Cdiff's time, not shown in the table, is basically the time taken to parse the two versions of a program. Averaged over all 10 programs, Cdiff's user, system, and elapsed times were 1.43, 0.43, and 2.04 seconds, respectively. Based on elapsed time, Cdiff processes a weighted average of 5,921 lines of code per second (6,508 based on user time only). This is 2 to 4 times faster than gcc running without producing output and without performing any optimization.

The final table, Table D, presents statistics related to the size of Δ. It includes the size of the SDG, the mean and median size of Δ over all runs for each program and the mean percent reduction for each of the ten programs. The average reduction over all 10 programs was 37.70%. Weighting the reduction of each program by its size produces a weighted average of 36.58%. This is an improvement over the 26% average reduction obtained from the smaller programs of Section 4.2. The improvement is consistent with the notion that larger programs should contain more than one thought, and thus produce smaller Δs.

A further breakdown for two of the largest programs is presented in Figure 5. Each pie graph groups together Δs by size into bands 10,000 vertices wide. The two programs represent two extremes in the effectiveness of computing `differences`. For acct, about 75% of the graph includes Δs containing less than half of the program and overall there are a variety of sizes present. In contrast, over 70% of the Δs for ed contain over 90,000 of the 98,479 vertices. It is clear from the figure that ed is a tightly knit program while acct is not.

Looking at graphs of the size of Δ for the 10 subject programs five patterns emerge. Figure 6 includes one representative graph for each pattern. The graphs show the percent reduction for all runs of the program sorted in ascending order.

Table D				
	SDG	Δ size		mean percent
program	size	mean	median	reduction
a2s	57,922	30,161	31,873	47.93%
acct	110,860	29,830	28,298	73.09%
barcode	195,62	13,209	15,646	32.48%
bc	70,502	64,909	67,990	7.93%
copia	46,130	29,404	22,295	36.26%
ed	98,479	79,823	90,721	18.94%
flex	53,418	29,660	37,875	44.48%
oracolo2	92,200	65,918	82,028	28.51%
prepro	95,757	68,321	85,225	28.65%
termutils	10,161	4,192	5,120	58.74%
average	65,499	41,543	46,707	37.70%

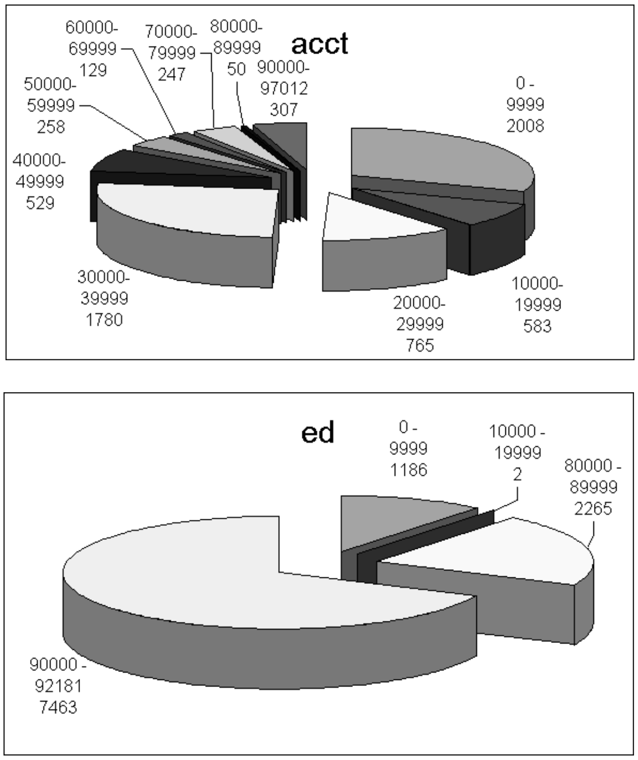

Figure 5. Summary of the size of Δ for two extreme examples. Each wedge counts the number of Δ's with size between x and $x + 10000$.

Four programs exhibit the pattern shown by oracolo2 in the top of Figure 6. They contain three horizontal bars. The first includes essentially all of the input program. The second bar includes 62 to 85 percent of the input program and the third includes only a very small portion of the program. The third bar mainly represents dead or forgotten code. This pattern is exhibited by tightly knit, single thought programs.

The second group includes more bands. In particular, a significant band where Δ includes only a small fraction of the original program (about 20% in the barcode example shown in Figure 6). The third and fourth groups show increased stratification and even the largest Δs don't include all of the program. Finally, the last group includes the widest variety of sizes. It also shows an inversion of the curve, which quickly rises to a reduction of over 60%.

Note that a graph of Δs size is a vertical reflection of the percent-reduction graph. It is also interesting to observe that by comparing the timing graphs in Figure <|times_figure|> with the size of oracolo2 in the top graph for Figure 6 the linear nature of the Δ computation can be seen.

Finally, as with any study, it is important to consider so called "threats to validity" in order to provide a perspective on the results. For example, external validity: do these results extend to other programs? To address external validity an ever increasing and varied selection of programs must be considered. Second, for the second group of programs, multiple versions were not available; thus, the study considers a single change at a time. While they are not expected to, multiple changes may produce different results. In limited informal observation this does not occur. There are several other potential threats that are the target of future study as discussed briefly in Section 7.

5. Uses of `differences`

The ability to extract semantically meaningful subprograms is clearly useful. Two example uses, testing and impact analysis, are briefly considered in this section.

5.1. Testing

`Differences` can be applied to testing both during initial development and, more effectively, during regression testing. First, to save time during initial development `differences` can be used in place of the entire program when testing certain aspects of the program. For testing techniques ranging from program walk throughs to the use of automatic test-case generation tools, testing the smaller program `differences` should reduce the cost of testing. A future study is planned to investigate the impact of testing `differences` in place of the entire program.

`Differences` is perhaps more useful in it's original proposed purpose: reducing the cost of regression testing [5]. The cost of regression testing can be reduced if unchanged parts of the program are not retested, and if old test cases and old test results are reused. `Differences` accomplishes the first of these while test-case selection algorithms [4, 7, 10, 14, 29] accomplish the second. As

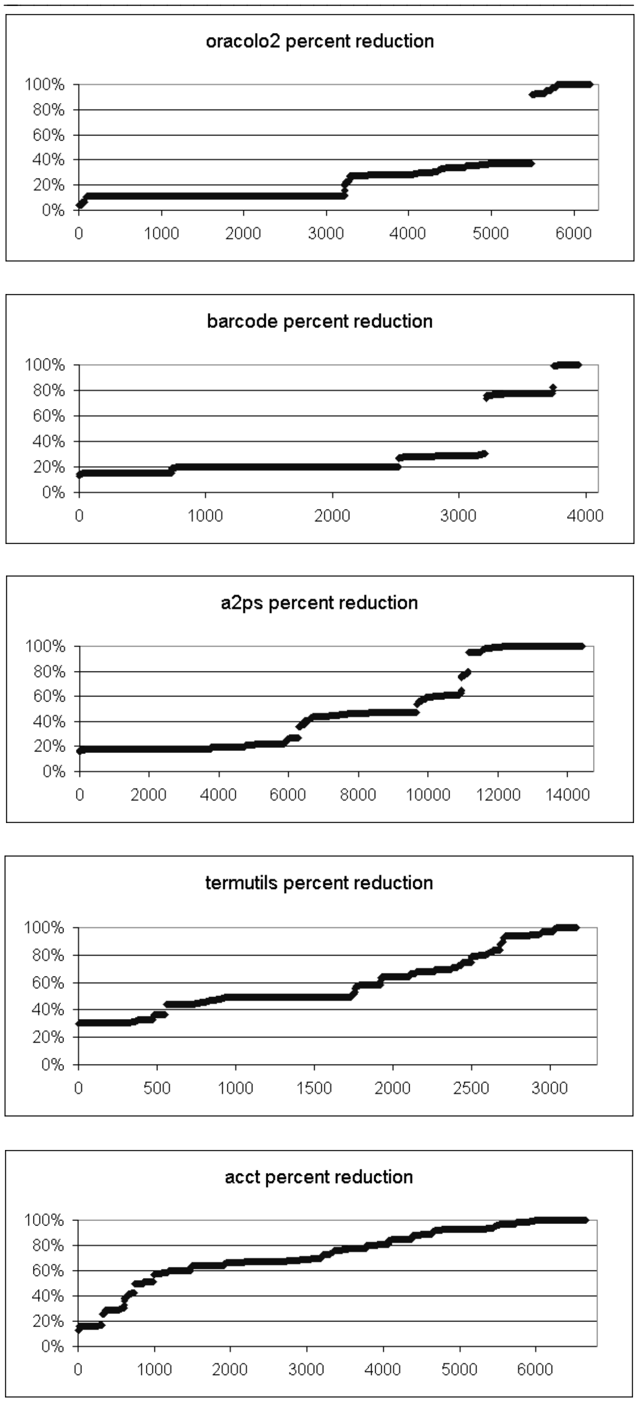

Figure 6. Percent reduction graphs for five representative programs.

these two approaches are complimentary, it should be possible to maximize savings by using both techniques.

The application of `differences` to regression testing is illustrated in the programs shown in Figures 2 and 7. Figure 2 illustrates the use of `differences` when maintenance calls for a program enhancement. In this example, testing `differences` in place of the entire enhanced program means not executing the code used to compute the `sum`. Note that test case selection, which can only decide whether or not to run a test case, but not which statements to execute, is incapable of being any assistance in this type of example.

The second illustration, shown in Figure 7, is a maintenance fix. The program, word_count, counts the number of lines, characters, and words in its input. In the example, the original version of the program contains an error: the variable `lines` was incorrectly initialized to 1. As a maintenance fix, `new` corrects this. Note that the change is in the program's initialization and thus affects all paths through the program. Many algorithms for test-case selection (*e.g.*, those studied in [30]) tend to include all test cases when a change occurs in the programs initialization [4]. While test-case selection has difficulty with changes made in the program's initialization, `differences` does not. As shown in Figure 7, `differences` includes only the computation of the number of lines. Thus, testing differences reduces the cost of testing. In this case `differences` is 40% smaller than `new`.

While in these two examples, test-case selection is not useful, in general it is. Furthermore, test-case selection and semantic differencing are complimentary techniques. Thus, the two should work well together.

Constructing `differences` can yield other benefits that offset its costs over and above savings in test execution. For example, simply displaying `differences` to the test designer helps them understand the portions of a program that are impacted.

5.2. Impact Analysis

In order to support the maintenance process, it is necessary to be able to accurately predict maintenance costs. This process, typically referred to as impact analysis, has many varied definitions [8, 25]. One broad definition of impact analysis is "the process of identifying software working products that may be affected by a proposed change" [8]. Other definitions also include the potential consequences of a change [1].

Dependence graphs are more useful in "code focused" impact analysis [8, 20]. Two example aspects of impact analysis in which Δ is useful are visual impact analysis and dealing with ripple effects.

Visual impact analysis is a software visualization technique that lets software maintainers judge the impact of proposed changes and plan maintenance accordingly. For

File Edit Queries Functions Go Window Help

```
/* modified version of word count */

static int isletter(char c);

main()
{
  char c;
  int inword, lines, words, characters;

  inword = 0;
  characters = 0;
  lines = 0;
  words = 0;

  while (scanf("%c", &c) == 1)
  {
    characters  = characters + 1;

    if (c == '\n')
      lines = lines + 1;

    if (isletter(c))
    {
      if (inword == 0)
      {
        words = words + 1;
        inword = 1;
      }
    }
    else
      inword = 0;

  }

  printf("%d ", lines);
  printf("%d ", words);
  printf("%d ", characters);
  printf("\n");
}
```

Figure 7. The word count program. `New` fixes the errant initialization of `lines` to 1. The statements that make up `differences` are highlighted in red by CodeSurfer and appear light gray in the figure. Statements representing affected points are underlined.

example, Hutchins and Gallagher describe a system that "visualizes impact analysis by displaying relationships among program components. It allows the software engineer to visually comprehend the relationship between decomposition slices" (a decomposition slice is essentially all the statements that affect the output of a particular variable) [18]. Since such statements can be viewed as having no dependence successors in the SDG, Δ essentially generalizes decomposition slicing (computing Δ on all outputs of a variable returns the decomposition slice).

The graphs from Section 4 provide a similar visualization: The slope and general shape of the curve predict the expected difficulty of making a change. Highlighting specific points in these graphs that correspond to the locations of a proposed change would provide a more specific indication of the complexity of the change. In comparison, the Hutchins and Gallagher visualization is considerably more difficult to read than the graphs of Section 4, but once understood contains more information. The two views are complimentary.

Ripple effects are a common problem faced by developers in an evolutionary environment: A seemingly small change can ripple throughout the system to have major unintended impacts elsewhere in the program. The operator Δ and it's underlying slicing operators can be used to help identify ripple effects. First, the set of affected points contains all the points potentially affected by a change. Clearly unexpected affected points are an indication of potential unexpected ripples. Second, `differences` can be extracted and used to plan a modification. This allows a maintainer to focus his or her attention on relevant code.

6. Related Work

Perhaps owing to its inherent complexities, semantic differencing is an under-studied area. This section briefly considers two algorithms for semantic differencing and one for syntactic differencing. There is too wide a variety of work related to the uses discussed in Section 5 to do them justice here. Therefore work related to regression testing [4, 10, 22, 14, 28, 26, 27, 13, 3, 7, 19], empirical studies of techniques for regression testing cost reduction [12, 34, 30, 31], and impact analysis [1, 2, 32, 21, 20, 18] are not addressed.

The first semantic differencing algorithm considered computes semantic differences of single procedure programs [15]. This algorithm is based on a related intermediate form called the *program representation graph*, which combines aspects of program dependence graphs and single assignment form [9]. It is interesting to note that when restricted to intraprocedural programs, this algorithm is more precise than the one presented in Section 2. However, attempts to extend this algorithm to multiple procedures have been unsuccessful. The second algorithm performs interprocedural semantic differencing [5, 7]. This algorithm is the basis for the implementation described in Section 4. Both of these algorithms had their genesis in work on semantics based program integration [6, 16, 36].

Finally, a recent study by Vokolos and Frankl uses textual differencing to select tests [34]. In comparison with semantic differencing, this approach is efficient (testing 6,000 lines of code in about 5 seconds) and has a straightforward implementation. What is given up for this efficiency is a loss of precision. In their study, this loss was minimal. It is interesting to note that while the hardware was not identical, this speed can be compared with the 5921 lines per second of Cdiff and the 47,270 vertices per second of the C implementation.

7. Summary and Future Work

Program `differences` captures the semantic differences between two programs and thus is clearly useful in many software engineering activities. This paper describes an implementation of `differences` for the C language, and investigates the size reduction obtained on a range of sample programs. A series of experiments with the implementation illustrate the effectiveness and efficiency of the algorithm. For example, with the mid-sized programs studied, `differences` reduced the portion of the program that must be considered during regression testing by an average of 37 percent.

Future work includes several experiments and extensions. The first experiment is designed to determine if computing and then testing `differences` is truly faster than simply testing the modified program. Work on this experiment has begun, and preliminary data indicate that a statistically significant improvement is achieved using `differences`. A follow-on study will combine `differences` with a test case selection algorithm. Finally, it would be useful to extend `differences` into the other parts of the testing infrastructure (*e.g.*, test scripts).

Acknowledgments
GrammaTech provided *CodeSurfer*^{tm} upon which the implementation is based. Siemens Corporate Research provided some of the test programs. Alberto Pasquini provided the programs oracolo2, prepro, and copia. Keith Gallagher noted the relation to impact analysis. Finally, the anonymous reviewers provided useful input.

References

1. Arnold, R.S. and Bohner, S.A., "Impact analysis - towards a framework for comparison," *Proceedings of the international conference Software Maintenance, 1993.*, pp. 292-301 (1993).
2. Barros, S., Bodhuin, Th., Escudie, A., Queille, J.P., and Voidrot, J.F., "Supporting impact analysis: a semi-automated technique and associated tool," *Proceedings of the international conference Software Maintenance, 1995.*, pp. 42-51 (1995).
3. Bates, S. and Horwitz, S., "Incremental program testing using program dependence graphs," in *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages,* (Charleston, SC, January 10-13, 1993), ACM, New York, NY (1993).
4. Benedusi, P., Cimitile, A., and De Carlini, U., "Post-maintenance testing based on path change analysis," pp. 352-368 in *Proceedings of the IEEE Conference on Software Maintenance,* (Phoenix, Arizona, Oct, 1988), IEEE Computer Society, Washington, DC (1988).
5. 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).
6. 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).
7. Binkley, D., "Semantics guided regression test cost reduction," *IEEE Transactions on Software Engineering* **23**(8) pp. 489-516 IEEE Computer Society, (August 1997).
8. Cimitile, A., Fasolino, A.R., and Visaggio, G., "A software model for impact analysis: a validation experiment," *Proceedings of the sixth working conference on reverse engineering, 1999*, pp. 212-222 (1999).
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K., "An efficient method of computing static single assignment form," pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
10. Fischer, K.F., Raji, F., and Chruscicki, A., "A methodology for re-testing modified software," pp. B6.3.1-6 in *IEEE National Telecommunications Conference Proceedings*, (Nov. 1981).
11. Grammatech Inc., *http://www.grammatech.com/ products/codesurfer/codesurfer.html*. 1999.
12. Graves, T. L.,, Harrold, M.J., Kim, J.-M., Porter, A., and Rothermel, G., "An empirical study of regression test selection techniques," *The 20th International Conference on Software Engineering*, pp. 188-197 (April 1998).
13. Gupta, R., Harrold, M.J., and Soffa, M.L., "An approach to regression testing using slicing," pp. 299-308 in *Proceedings of the IEEE Conference on Software Maintenance,* (Orlando, Florida, Nov, 1992), IEEE Computer Society, Washington, DC (1992).
14. Harrold, M.J. and Soffa, M.L., "An incremental approach to unit testing during maintenance," pp. 362-367 in *Proceedings of the IEEE Conference on Software Maintenance,* (Phoenix, Arizona, Oct, 1988), IEEE Computer Society, Washington, DC (1988).
15. 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).

16. 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).

17. 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).

18. Hutchins, M. and Gallagher, K., “Improving visual impact analysis,” *Proceedings of the international conference Software Maintenance, 1999.*, pp. 294-303 (1998).

19. Kamkar, M., Fritzson, P., and Shahmehri, N., “Interprocedural dynamic slicing applied to interprocedural data flow testing,” pp. 368-395 in *Proceedings of the IEEE Conference on Software Maintenance,* (Montreal, Quebec, Canada, Sep, 1993), IEEE Computer Society, Washington, DC (1993).

20. Kim, K.H., Park, J.N., and Yoon, Y.I., “A graph of change impact analysis for distributed object-oriented software,” *Proceedings of the 1999 IEEE internationsl conference on fuzzy systems. (Seoul Korea).*, (august 1999).

21. McCrickard, D.S. and Abowd, G.D., “Assessing the impact of changes at the architectural level: a case study on graphical debuggers,” *Proceedings of the international conference Software Maintenance, 1996.*, pp. 58-67 (1996).

22. Ostrand, T.J. and Weyuker, E.J., “Using data flow analysis for regression testing,” in *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference,* (Portland, Oregon, September 19-20, 1988), ACM, New York, NY (1988).

23. Ottenstein, K.J. and Ottenstein, L.M., “The program dependence graph in a software development environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Pittsburgh, PA, Apr. 23-25, 1984)*, ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

24. 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).

25. Queille, J.P., Voidrot, J.F., Wilde, N., and Munro, M., “The impact analysis task in software maintenance: a model and a case study,” *Proceedings of the international conference Software Maintenance, 1994.*, pp. 234-242 (1994).

26. Rothermel, G. and Harrold, M.J., “A safe, efficient algorithm for regression test selection,” *Proceedings of the IEEE Conference on Software Maintenance,* (Montreal, Quebec), pp. 358-367 IEEE Computer Society, (September 1993).

27. Rothermel, G. and Harrold, M.J., “Selecting tests and identifying test coverage requirements for modified software,” *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* Seattle, Washington), pp. 169-84 (August 1994).

28. Rothermel, G. and Harrold, M.J., “Analyzing regression test selection techniques,” *IEEE Transactions on Software Engineering* **22**(8)IEEE Computer Society, (1996).

29. Rothermel, G. and Harrold, M.J., “A safe, efficient regression test set selection technique,” *ACM Transactions on Software Engineering and Methodology* **6**(2,) pp. 173-210 (April 1997).

30. Rothermel, G. and Harrold, M.J., “Empirical studies of a safe regression test selection technique,” *IEEE Transactions on Software Engineering* **24**(6) pp. 401-419 (June 1998).

31. Rothermel, G., Untch, R., Chu, C., and Harrold, M.J., “Test case prioritization: an empirical study,” pp. 179-188 in *IEEE International Conference on Software Maintenance,* (Oxford, UK, September, 1999), IEEE Computer Society, Washington, DC (1999).

32. Sneed, H.M., “Estimating the costs of software maintenance tasks,” *Proceedings of the international conference Software Maintenance, 1995.*, pp. 168-181 (1995).

33. Tip, F., “A survey of program slicing techniques,” *Journal of Programming Languages* **3**(3) pp. 121-189 (September 1995).

34. Vokolos, F. and Frankl, P., “Empirical evaluation of the textual differencing regression testing technique,” pp. 44-53 in *IEEE International Conference on Software Maintenance,* (Washington, DC, Nov, 1998), IEEE Computer Society, Washington, DC (1998).

35. Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).

36. Yang, W., Horwitz, S., and Reps, T., “A program integration algorithm that accommodates semantics-preserving transformations,” *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments,* (Irvine, CA, December 3-5, 1990)*, ACM SIGSOFT Software Engineering Notes*, (1990).

37. Yang, W., “Identifying syntactic differences between two programs,” *Software—Practice and Experience* **21**(7)(July 1991).