Directed Test Suite Augmentation: Techniques and Tradeoffs

Zhihong Xu[†], Yunho Kim[*], Moonzoo Kim[*], Gregg Rothermel[†], Myra B. Cohen[†]

[†]Department of Computer Science and Engineering
University of Nebraska - Lincoln
{zxu,grother,myra}@cse.unl.edu

[*]Computer Science Department
Korea Advanced Institute of Science and Technology
kimyunho@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

ABSTRACT

Test suite augmentation techniques are used in regression testing to identify code elements affected by changes and to generate test cases to cover those elements. Our preliminary work suggests that several factors influence the cost and effectiveness of test suite augmentation techniques. These include the order in which affected elements are considered while generating test cases, the manner in which existing regression test cases and newly generated test cases are used, and the algorithm used to generate test cases. In this work, we present the results of an empirical study examining these factors, considering two test case generation algorithms (concolic and genetic). The results of our experiment show that the primary factor affecting augmentation is the test case generation algorithm utilized; this affects both cost and effectiveness. The manner in which existing and newly generated test cases are utilized also has a substantial effect on efficiency but a lesser effect on effectiveness. The order in which affected elements are considered turns out to have relatively few effects when using concolic test case generation, but more substantial effects when using genetic test case generation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

1. INTRODUCTION

Software engineers use regression testing to validate software as it evolves. To do this cost-effectively, they often begin by running existing test cases. Existing test cases, however, may not be adequate to validate the code or system behaviors that are present in a new version of a system. *Test suite augmentation techniques* (e.g., [2, 32, 42]) address this problem, by identifying where new test cases are needed and then creating them.

Despite the need for test suite augmentation, most research on regression testing has focused on using existing test cases. There has been research on approaches for *identifying affected elements* (code components potentially affected by changes) [2, 5, 20, 30, 32], but these approaches leave the task of generating new test cases to engineers. There has been research on automatically generating test cases given pre-supplied coverage goals (e.g., [13, 29, 34, 37]), but this research has not attempted to integrate the test case generation task with reuse of existing test cases for augmentation.

In principle, any test case generation technique could be used to generate test cases for a modified program. We believe, however, that test case generation techniques that leverage existing test cases hold the greatest promise where test suite augmentation is concerned. This is because existing test cases provide a rich source of data on potential inputs and code reachability, and existing test cases are naturally available as a starting point in the regression testing context. Further, recent research on test case generation has resulted in techniques that rely on dynamic test execution, and such techniques can naturally leverage existing test cases.

In prior work [42] we developed a *directed test suite augmentation technique*. The technique begins by using a regression test selection algorithm [31] to identify code affected by changes and existing test cases relevant to testing that code. The technique then uses the identified test cases to seed a concolic test case generation approach [34] to create test cases that execute the affected code. A case study shows that the approach improves both the efficiency of the technique and its ability to cover affected elements. Further work [41] examined a similar approach to augmentation using a genetic algorithm for test case generation.

While these initial results are encouraging, our attempts to create augmentation techniques show that several factors can potentially influence the cost and effectiveness of those techniques. Three factors in particular appear to be: (1) the order in which affected elements are considered while generating test cases, (2) the manner in which existing and newly generated test cases are used, and (3) the algorithm used to generate test cases.

To create effective test suite augmentation techniques we need to understand the influence of the foregoing factors. Based on such an understanding, we should be better able to create augmentation techniques that leverage test cases in a cost-effective manner. We have therefore designed and conducted a controlled experiment investigating these factors in the context of test suite augmentation. Our experiment considers concolic and genetic test case generation algorithms, two different orderings of affected elements, and two different manners of using existing test cases. We consider each relevant combination on these on four object programs, measuring the effectiveness of the approaches in terms of code coverage, and their costs in terms of the time required to perform augmentation.

The results of our experiment show that among the factors that we consider, the primary factor affecting augmentation is the algorithm utilized to generate test cases; this affects both augmentation cost and effectiveness. The manner in which existing and newly generated test cases are utilized also has a substantial effect on efficiency but a lesser effect on effectiveness. The order in which affected elements are considered turns out to have relatively few effects when using concolic test case generation, but more substantial effects when using genetic test case generation.

This work makes several contributions. (1) We provide a new formalized algorithm for performing augmentation using various test case generation algorithms and various settings of potentially influential factors. (2) We report results on the first controlled experiment considering test suite augmentation, and the first such experiment to compare different test case generation techniques in the augmentation context (genetic and concolic). (3) Our results provide additional evidence that directed test suite augmentation techniques can be effective. (4) Our results reveal factors that researchers and experimentalists should consider when attempting to create and study directed test suite augmentation techniques.

2. BACKGROUND

2.1 Test Suite Augmentation

Let P be a program, let P' be a modified version of P, and let T be a test suite for P. Regression testing is concerned with validating P'. To facilitate this, engineers often begin by reusing T, and a wide variety of approaches have been developed for rendering such reuse more cost-effective via regression test selection (RTS) (e.g., [28, 31]) and test case prioritization (e.g., [15, 23, 38]).

Test suite augmentation techniques, in contrast, are not concerned with reuse of T. Rather, they are concerned with the tasks of (1) *identifying affected elements* (portions of P' or its specification for which new test cases are needed), and then (2) *creating or guiding the creation of test cases that exercise these elements*.

Various algorithms have been proposed for identifying affected elements in software systems following changes. Some of these [6] operate on levels above the code such as on models or specifications, but most operate at the level of code, and in this paper we focus on these. Code level techniques [5, 20, 30] use various analyses, such as slicing on program dependence graphs, to select existing test cases that should be re-executed, while also identifying portions of the code that are related to changes and should be tested. However, these approaches do not provide methods for generating actual test cases to cover the identified code.

Four recent papers [2, 29, 32, 42] specifically address test suite augmentation. Two of these [2, 32] present an approach that combines dependence analysis and symbolic execution to identify chains of data and control dependencies that, if tested, are likely to exercise the effects of changes. A potential advantage of this approach is a fine-grained identification of affected elements; however, the papers present no specific algorithms for generating test cases. A third paper [29] presents an approach to program differencing using symbolic execution that can be used to identify affected elements more precisely than [2, 32], and yields constraints that can be input to a solver to generate test cases for those requirements. However, this approach is not integrated with reuse of existing test cases.

As mentioned in Section 1, the test suite augmentation approach that we presented in [42] integrates an RTS technique [31] with an adaptation of the concolic test case generation approach presented in [34]. This approach leverages test resources and data obtained from prior testing sessions to perform both the identification of coverage requirements and the generation of test cases to cover these. The augmentation approach presented in [41] operates similarly, but uses a genetic algorithm to generate test cases. Case studies of the approaches shows that both can be effective and efficient. Both of these studies, however, are small, and neither study compares multiple augmentation approaches. Further, while [41] describes potentially influencing factors it investigates only one.

2.2 Test Case Generation

While in practice test cases are often generated manually, there has been a great deal of research on techniques for automated test case generation. For example, there has been work on generating test cases from specifications, from formal models and by random or quasi-random selection of inputs (e.g., [8, 24, 27, 36]).

In this work we focus on code-based test case generation techniques, many of which have been investigated in prior work. Among these, several techniques (e.g., [9, 12, 19]) use symbolic execution to find the constraints, in terms of input variables, that must be satisfied in order to execute a target path, and attempt to solve this system of constraints to obtain a test case for that path.

While the foregoing test case generation techniques are static, other techniques make use of dynamic information. Execution-oriented techniques [22] incorporate dynamic execution information into the search for inputs, using function minimization to solve subgoals that contribute toward an intended coverage goal. Goal-oriented techniques [17] also use function minimization to solve subgoals leading toward an intended coverage goal; however, they focus on the final goal rather than on a specific path, concentrating on executions that can be determined through analysis to possibly influence progress toward that goal.

Several test case generation techniques use evolutionary or search-based approaches (e.g., [4, 13, 26, 37]) such as genetic algorithms, tabu search, and simulated annealing to generate test cases. Other work [7, 10, 18, 33, 34] combines concrete and symbolic test execution to generate test inputs. This second approach is known as *concolic testing* or *dynamic symbolic execution*, and has proven useful for generating test cases for C and Java programs. The approach has been extended to generate test data for database applications [16] and for Web applications using PHP [3, 40].

3. AUGMENTATION TECHNIQUES

We now describe the augmentation techniques that we consider.

3.1 Augmentation Basics

3.1.1 Coverage Criterion

We are interested in code-based augmentation techniques, and these typically involve specific code coverage criteria. In this work, we focus on code coverage at the level of *branches*; that is, outcomes of predicate statements. While stronger than statement coverage, branch coverage is more tractable than criteria such as path coverage, and more likely to scale to larger systems.

3.1.2 Identifying Affected Elements

As noted in Section 1, test suite augmentation consists of two tasks, identifying affected elements and creating test cases that exercise these elements. In this work the factors we are studying concern the second of these tasks; thus, we choose a typical and practical approach for performing the first. Given program P and its test suite T, and modified version P' of P, to identify affected elements in P' we execute the test cases in T on P' and measure their branch coverage. Any branch in P' that is not covered is an affected element. This approach corresponds to the common "retest-all" regression testing process in which existing test cases are executed on P' first, and then, augmentation is performed where needed.

3.1.3 Ordering Affected Elements

Our augmentation techniques operate on lists of affected elements, and we believe that the order in which these elements are considered can affect the techniques. In this work, we investigate the use of a depth-first order of affected elements.

The depth-first order (DFO) of nodes in a graph is the reverse of the order in which nodes are last visited in a preorder traversal of the graph [1]. In dataflow analysis, DFO causes nodes that are "earlier" in control flow to be considered prior to those that follow them, and can speed up the convergence of an analysis. We conjecture that by considering affected elements in this order, we may be able to speed up the process of generating test cases, because test cases generated for elements earlier in flow may incidentally cover elements occurring later in flow, obviating the need to consider those later elements again.

In our case we construct the DFO in terms of branches, and on a program's interprocedural control flow graph (ICFG). We first build the ICFG, then we traverse the ICFG recording the branches that we visit (both forward and while backtracking). This recorded information lets us calculate the reverse of the order in which branches are last visited. Finally, we filter out branches that were not designated as affected to obtain our ordered list of affected elements.

3.1.4 Main Augmentation Algorithm

Algorithm 1 controls the augmentation process, beginning with an initial set of existing test cases, TC, an ordered set of affected elements (target branches), $B_{aff_{ini}}$, and an iteration limit n_{iter}. The algorithm assigns $B_{aff_{ini}}$ to B_{aff} (line 1), which henceforth contains a set of affected elements still needing to be covered. The main loop (lines 3-16) continues until we can no longer increase coverage (which may result due to reaching the iteration limit in the test case generation routines). Within this loop, for each branch $b_t \in B_{aff}$, if b_t is not covered we call a test case generation algorithm to generate test cases (line 7). If the algorithm successfully generates and returns new test cases this means that at least some new coverage has been achieved in the program (although b_t may or may not have been covered in the process).

```
Input: set of existing test cases $TC$, ordered set of affected
       elements $B_{aff_{ini}}$, and an iteration limit $n_{iter}$
Output: $TC$ augmented with new test cases
1  $B_{aff} = B_{aff_{ini}}$
2  $NewCoverage$=true;
3  while $NewCoverage$ do
4      $NewCoverage$=false
5      foreach $b_t \in B_{aff}$ do
6          if $NotCovered$ then
7              $NewTests$ =AUGMENT($TC, B_{aff}, b_t, n_{iter}$)
8              if $NewTests$ !=$Empty$ then
9                  $NewCoverage$=true
10             end
11             if $UseNew$ then
12                 $TC$=$NewTests \cup TC$
13             end
14         end
15     end
16 end
```

Algorithm 1: Main Augmentation Algorithm

To accommodate our other factor of concern — the manner in which existing and new test cases are used — we allow for the possibility of adding the newly generated test cases back into our set of available test cases. If the boolean flag $UseNew$ is set to true, this causes the algorithm to combine the newly generated test cases with the original test cases (lines 11-12), and then this newly formed TC is used for the next iteration of our algorithm.

We next describe two different test case generation algorithms that can be invoked at line 7 to generate new test cases.

3.2 Genetic Test Suite Augmentation

Genetic algorithms for structural test case generation begin with an initial (often randomly generated) test data population and evolve the population toward targets that can be blocks, branches or paths in a program [25, 35, 39]. To apply such an algorithm to a program, the test inputs must be represented in the form of a chromosome, and a fitness function must be provided that defines how well a chromosome satisfies the intended goal. The algorithm proceeds iteratively by evaluating all chromosomes in the population and then selecting a subset of the fittest to mate. These are combined in a crossover stage where information from one half of the chromosomes is exchanged with information from the other half to generate a new population. A small percentage of chromosomes in the new population are mutated to add diversity back into the population. This concludes a single generation of the algorithm. The process is repeated until a stopping criterion has been met.

Algorithm 2 describes the genetic algorithm used in our experiment. The algorithm accepts four parameters: a set of test cases TC, a set of affected elements B_{aff}, an uncovered target branch b_t, and an iteration limit n_{iter}. The algorithm returns a set of new test cases NTC, consisting of all test cases generated that covered any previously uncovered branches in P.

Instead of using random test cases to form an initial population, we take advantage of existing test cases to seed the population. We run this algorithm for each target branch b_t. As the starting population, we select all of the test cases reaching method m_{b_t}, the method that contains b_t; this determines the population size.

```
Input: a set of test cases $TC$, a set of affected elements
       $B_{aff}$, an uncovered target branch $b_t \in B_{aff}$, and
       an iteration limit $n_{iter}$
Output: a set of new test cases $NTC$
1  $TC_{cur} = TC$ // set of current target test cases
2  $NTC = \emptyset$ // set of new test cases generated
3  $TC_{b_t}$ = {test cases in $TC_{cur}$ that reach method $m_{b_t}$, the
   method containing $b_t$}
4  $Population = TC_{b_t}$
5  $i = 0$
6  repeat
7      $Fitness$=CalculateFitness($Population$)
8      $Population$=Select($Population$, $Fitness$)
9      $Population$=Crossover($Population$)
10     $Population$=Mutate($Population$)
11     $i = i + 1$
12     foreach $tc \in Population$ do
13         Execute ($tc$)
14         if $tc$ covers new branches in $B_{aff}$ then
15             Update $B_{aff}$
16             $NTC = NTC \cup$ {tc}
17         end
18     end
19 until $i \geq n_{iter}$ or $b_t$ is covered;
20 return $NTC$
```

Algorithm 2: GENETIC-AUGMENT algorithm

The algorithm repeats for a number of generations (set by the variable n_{iter}) or until b_t is covered. The first step (line 7) is to calculate the fitness of all test cases in the population. Since the fitness of a test case depends on its relationship to the branch we are trying to cover, calculating the fitness requires that we run the test case. (For test cases provided initially we can use coverage information obtained while performing the prior execution of TC, which in our case occurred in conjunction with determining affected elements.) Next a selection is performed (line 8), which orders and chooses the best half of the chromosomes to use in the next step. This population is divided into two halves (retaining the ranking) and the first chromosome in the first half is mated with the first chromosome in the second half and this continues until all have been mated. Next (line 10) a small percentage of the population is mutated, after which all test cases in the current population are executed. If b_t is covered or the iteration limit is met we are finished (line 19), otherwise we iterate.

3.3 Concolic Test Suite Augmentation

Concolic testing (concolic execution) [7, 18, 34] concretely executes a program while carrying along a symbolic state and simultaneously performing symbolic execution of the path that is being executed. It then uses the symbolic path constraint gathered along the way to generate new inputs that will drive the program along a different path on a subsequent iteration, by negating a predicate in the path constraint. In this way, concrete execution guides the symbolic execution and replaces complex symbolic expressions with concrete values when needed to mitigate the incompleteness of the constraint solvers [34]. Conversely, symbolic execution helps to generate concrete inputs for the next execution to increase coverage in the concrete execution scope.

In the traditional application of concolic testing, test case reuse is not considered, and the focus of test generation is on path coverage. First, a random input is applied to the program and the algorithm collects the path condition for this execution. Next, the algorithm negates the last predicate in this path condition and obtains a new path condition. Calling a constraint solver on this path condition yields a new input, and a new iteration then commences, in which the algorithm again attempts to negate the last predicate. If the algorithm discovers that a path condition has been encountered before, it ignores it and negates the second-to-last predicate. This process continues until no more new path conditions can be generated. Ideally, the end result of the process is a set of test cases that cover all paths.

In this work, we alter the foregoing approach to function in the context of the main augmentation algorithm presented in Section 3.1.4; this includes leveraging existing test cases and operating on an ordered list of affected elements, at the level of branch coverage.

We use the following notation:

- $CFG_P = (N_P, E_P)$ is a control flow graph of a target program P where N_P is a set of nodes (statements in P) and E_P is a set of edges (branches in P) between N_P.
- A *path condition pc* of a target program P is a conjunction $b_{i_1} \wedge b_{i_2} \wedge ...b_{i_n}$ where $b_{i_1}, ...b_{i_n}$ are edges in E_P and executed in order. Note that n can be larger than $|E_P|$, since one branch in a loop body of P may be executed multiple times (i.e., it is possible that $b_{i_k} = b_{i_l}$ for $k \neq l$).
- $DelNeg(pc, j)$ generates a new path condition from a path condition pc by negating a branch occurring at the jth position in pc and removing all subsequent branches. For example, $DelNeg(b_{i_1} \wedge b_{i_2} \wedge b_{i_3}, 2) = b_{i_1} \wedge \neg b_{i_2}$.
- $\overline{b}$ is a paired branch of a branch b (i.e., if b is a `then` branch, $\overline{b}$ is the `else` branch).
- $LastPos(b, pc)$ returns a last position j of a branch b_{i_j} in a path condition pc where $b = b_{i_j}$ (i.e., $\forall j < k \leq n. b_{i_k} \neq b$).
- $Solve(pc)$ returns a test case satisfying the path condition pc if pc is satisfiable; UNSAT otherwise.

Algorithm 3 describes our concolic augmentation algorithm. The algorithm accepts the same four parameters accepted by the genetic algorithm, and returns a set NTC of new test cases. Lines 4-23 detail the main procedure of the algorithm.

```
Input: a set of test cases $TC$ , a set of affected elements
       $B_{aff}$, an uncovered target branch $b_t \in B_{aff}$, and
       an iteration limit $n_{iter}$
Output: a set of new test cases $NTC$
1  $TC_{cur} = TC$ // a set of the current target test cases
2  $NTC = \emptyset$ // a set of all new test cases generated
3  repeat
4      $NTC_{cur} = \emptyset$ // a set of newly generated test cases in
       the current execution of line 3 to line 23
5      $TC_{\overline{b_t}} = \{$ all test cases in $TC_{cur}$ that reach $\overline{b_t}$ $\}$
6      if $TC_{\overline{b_t}} = \emptyset$ then
7          return $\emptyset$
8      end
9      $PC_{\overline{b_t}} = \{$ path conditions obtained from executing
       test cases in $TC_{\overline{b_t}}\}$
10     foreach $pc \in PC_{\overline{b_t}}$ do
11         foreach $i = LastPos(\overline{b_t}, pc)$ to $i - n_{iter}+1$ do
12             if $i > 0$ then
13                 $pc' = DelNeg(pc, i)$
14                 $tc_{new} = Solve(pc')$
15                 if $tc_{new} \neq$ UNSAT and $tc_{new}$ covers
                   uncovered branches in $B_{aff}$ then
16                     Update $B_{aff}$
17                     $NTC_{cur} = NTC_{cur} \cup \{tc_{new}\}$
18                 end
19             end
20         end
21     end
22     $TC_{cur} = NTC_{cur}$
23     $NTC = NTC \cup NTC_{cur}$
24 until $NTC_{cur} = \emptyset$;
25 return $NTC$
```

Algorithm 3: CONCOLIC−AUGMENT algorithm

Initially, the current target test cases TC_{cur} (from which new test cases are generated) are the old test cases TC (line 1) and NTC is empty (line 2). The start of the main procedure resets the set of newly generated test cases NTC_{cur} (line 4) and selects test cases that can reach $\overline{b_t}$ (the paired branch of b_t) from among the current target test cases TC_{cur} (line 5). If there are no such test cases, the algorithm terminates (lines 6-8). If there are such test cases, the algorithm obtains path conditions by executing the target program with selected test cases (line 9). From each obtained path condition pc, the algorithm generates n_{iter} new path conditions as follows. Suppose the last occurrence of $\overline{b_t}$ is located in the mth branch of pc. Then, the algorithm generates n_{iter} new path conditions (lines 11-19) by negating $b_{i_m}, b_{i_{m-1}}, ..., b_{i_{m-n_{iter}+1}}$ and removing all following branches in pc, respectively (line 13). If a newly generated path condition pc' has a solution tc_{new} (a new test case) (line 14) and tc_{new} covers uncovered branches in B_{aff} (line 15), B_{aff} is updated to reflect the new status of coverage (line 16), and tc_{new} is added to the set of newly generated test cases NTC_{cur} (line 17).

Note that the iteration limit n_{iter} parameter is a "tuning" parameter that determines how far back in a path condition the augmentation approach will go, and in turn can affect both the efficiency and the effectiveness of the approach.

4. EMPIRICAL STUDY

Our goal is to investigate the two augmentation techniques considered, focusing on the factors we have discussed. We thus pose the following research questions.

RQ1: How does the order of consideration of affected elements affect augmentation techniques?

RQ2: How does the use of existing and newly generated test cases affect augmentation techniques?

RQ3: How do genetic and concolic test case generation techniques differ in the augmentation context?

4.1 Objects of Analysis

To facilitate technique comparisons, programs must be suitable for use by both implementations. Also, programs must be provided with test suites that need to be augmented. To select appropriate objects we examined C programs available in the SIR repository [14]. We selected four programs (see Table 1), each of which is available with a large "universe" of test cases, representing test cases that could have been created by engineers in practice for these programs to achieve requirements and code coverage [21].

The object programs that we selected do not have actual sequential versions that can be used to model situations in which evolution renders augmentation necessary. We were able, however, to define a process by which a large number of test suites that need augmenting, and that possess a wide range of sizes and levels of coverage adequacy, could be created for the given object program versions. This lets us model a situation where the given versions have evolved rendering prior test suites inadequate, and require augmentation.

To create such test suites we did the following. First, for each object program P we used a greedy algorithm to sample P's associated test universe U, to create test suites that were capable of covering all the branches coverable by test cases in U. Next, we measured the minimum size T_{min} and maximum size T_{max} for these suites. We then randomly chose a number n such that $T_{min} \leq n \leq T_{max}$, and randomly selected n test cases from U to create a test suite, A. We measured the coverage achieved by A on P, and if A was coverage-adequate for P we discarded it. We repeated this step until 100 non-coverage-adequate test suites had been created. Statistics on the sizes and coverages of these test suites are given in Table 2.

Table 1: Experiment Objects

Program	*Functions*	*LOC*	*Branches*	*Test Cases*
`printtok1`	21	402	174	3052
`printtok2`	20	483	186	3080
`replace`	21	516	206	3174
`tcas`	8	138	76	1608

Table 2: Branch Coverage and Sizes of Initial Test Suites

Program	*Branch Coverage*			*Test Suite Size*		
	Avg	Min	Max	Avg	Min	Max
`printtok1`	141.1	122	155	21.8	18	25
`printtok2`	164.2	147	176	23.1	17	29
`replace`	171.7	141	181	23.1	19	28
`tcas`	61.7	44	69	13.0	11	15

4.2 Variables and Measures

4.2.1 Independent Variables

Our experiment manipulated three independent variables:

IV1: Order in which affected elements are considered. As orders of affected elements, we use the depth-first order described in Section 3.2, and a baseline approach that orders affected elements randomly.

IV2: Manner in which existing and new test cases are reused. We consider two approaches to reusing test cases; namely, the approach in which a test case generation algorithm attempts to utilize only existing test cases, and the approach in which it uses existing along with newly generated test cases.

IV3: Test case generation technique. We consider two test case generation techniques; namely, the genetic and concolic techniques described in Sections 3.2 and 3.3, respectively.

4.2.2 Dependent Variables and Measures

We wish to measure both the effectiveness and the cost of augmentation techniques under each combination of potentially affecting factors. To do this we selected two variables and measures.

DV1: Effectiveness in terms of coverage. The test case augmentation techniques that we consider are intended to work with existing test suites to achieve higher levels of coverage in a modified program P'. To measure the effectiveness of our techniques, we track the number of branches in P' that can be covered by each augmented test suite.

DV2: Cost in terms of time. To track the cost of augmentation, for each application of an augmentation technique we measure the wall clock time required to apply the technique.

4.3 Experiment Setup

Several steps had to be followed to establish the experiment setup needed to conduct our experiment.

4.3.1 Extended Programs

To implement our concolic test case generation technique we created a tool based on CREST [11]. CREST transforms a program's source code into an "extended" version in which each original conditional statement with a compound Boolean condition is transformed into multiple conditional statements with atomic conditions without Boolean connectives (i.e., `if(`b_1 `&&` b_2`) f()` is transformed into `if(`b_1`) { if(`b_2`) f()}`). To facilitate fair comparisons between concolic and genetic algorithms, however, we cannot apply the former to extended programs and the latter to non-extended programs. We thus opted to create extended versions of all four programs, and apply both algorithms to those versions.

4.3.2 Iteration Limits

Genetic algorithms iteratively generate test cases, and an iteration limit governs the stopping point for this activity. Similarly, the concolic approach that we use employs an iteration limit that governs the maximum number of path conditions that should be solved to generate useful test cases.

These iteration limits can affect both the effectiveness and the cost of the algorithms. Thus, we cannot run experiments with just one iteration limit per approach, because this would result in a case where our comparisons might reflect iteration limits rather than differences in techniques. For this reason, we chose multiple iteration limits for each test case generation approach, using 1-3-5-7-9 for concolic, and 5-10-15-20-25 for genetic. (The different numbers are due to the different meanings of iterations across the two algorithms, as explained in Sections 3.2 and 3.3.)

4.3.3 Technique Tuning

Genetic algorithms must be tuned to the object programs on which they are to be run. This does not present a problem in a test suite augmentation setting, because tuning can be performed on early system versions, and then the resulting tuned algorithms can be utilized on subsequent versions. For this study, we tuned our genetic algorithms by applying them directly to the extended object programs absent any existing suites.

4.4 Experiment Operation

Given our independent variables, an individual augmentation technique consists of a triple, (G,A,M), where G is one of the two test case generation techniques (Genetic or Concolic), A is one of two affected element orders (Random or Depth-first), and M is one of the two test case reuse approaches (Old test cases or New+old test cases). An individual *augmentation technique application* consists of an augmentation technique applied at an iteration limit L, where L is one of our five values.

Our experiment thus employs eight augmentation techniques and 40 augmentation technique applications. Each of these is applied to each of our four object programs for each of the 100 test suites that we created for that program. This results in 16,000 augmentation technique applications, for each of which we collect our dependent variables to obtain the data sets needed for our analysis.

Our experiments were run on a Linux box with an Intel Core2duo E8400 at 3.6GHz and with 16GB RAM, running Fedora 9 as an OS.

4.5 Threats to Validity

The primary threat to *external validity* for this study involves the representativeness of our object programs and test suites. We have examined only four relatively small C programs, and other objects may exhibit different cost-benefit tradeoffs. Furthermore, our programs are chosen to allow application of both genetic and concolic testing, and thus, do not reveal cases in which program characteristics might disable one but not the other of these approaches. A second threat to external validity pertains to our algorithms; we have utilized only one variant of a genetic test case generation algorithm, and one variant of a concolic testing algorithm, and we have applied both to extended versions of the object programs, where the genetic approach does not require this and might function differently on the original source code. Subsequent studies are needed to determine the extent to which our results generalize.

The primary threat to *internal validity* is possible faults in the implementation of the algorithms and in tools we use to perform evaluation. We controlled for this threat through extensive functional testing of our tools. A second threat involves inconsistent decisions and practices in the implementation of the techniques studied; for example, variation in the efficiency of implementations of techniques could bias data collected.

Where *construct validity* is concerned, there are other metrics that could be pertinent to the effects studied. In particular, our measurements of cost consider only technique run-time, and omit costs related to the time spent by engineers employing the approaches. Our time measurements also suffer from the potential biases detailed under internal validity, given the inherent difficulty of obtaining an efficient technique prototype.

5. RESULTS AND ANALYSIS

As an initial overview of the data, Tables 3, 4, 5 and 6 present the average coverage and cost values obtained per program, across all test suites, for each iteration level, for each combination of order of affected elements and test reuse approach. Each table presents results for concolic and genetic techniques under one combination.

We now present and analyze our data with respect to our three research questions, in turn.

5.1 RQ1: Order of Affected Elements

Our first research question pertains to the effects of using different orders of affected elements; in this case, depth-first order versus random. Table 7 presents data relevant to this question. The table presents results per program, with coverage results in the left half and cost results in the right half. Column headers use mnemonics to indicate techniques: GDO corresponds to (Genetic, DFO, Old), GDN to (Genetic, DFO, New+old), GRO to (Genetic, Random, Old), GRN to (Genetic, Random, New+old), CDO to (Concolic, DFO, Old), CDN to (Concolic, DFO, New+old), CRO to (Concolic, Random, Old), and CRN to (Concolic, Random, New+old). Individual columns correspond to techniques compared, thus, column 2, with header "GDO vs GRO", compares (Genetic, DFO, Old) to (Genetic, Random, Old), In each column, then, the only source of variance between the techniques compared is the order.

Each entry in the table summarizes the differences observed between the two techniques, for each of the five iteration limits, with "D" indicating that the technique using depth-first order exhibited the greater mean coverage or cost value, "R" indicating that the technique using random order exhibited the greater mean coverage or cost value, and "=" indicating that techniques exhibited equal mean coverage. For example, for `tcas`, comparing GDO and GRO, the table contains "D D = R R", indicating that at the lowest two iteration levels depth-first order produced better coverage, at the third level the orders produced equal coverage, and at the upper two levels random order produced better coverage.

For each pair of techniques compared, for each iteration limit L, we applied a *t-test* to the coverage (cost) data obtained across all test suites augmented, to determine whether there is a statistically significant difference between the two techniques at iteration limit L, using $\alpha = 0.05$ as the confidence level. In the table, bold-italicized fonts indicate statistically significant differences. For example, for `printtok1`, comparing GDO and GRO, the only statistically significant difference between techniques occurred at iteration level 15. It is these statistical differences that we focus on with respect to our research question.

We begin by considering the results for the genetic algorithm. Where coverage is concerned, no clear advantage resides in either test case order, and results are relatively similar in the cases where old or new and old test cases are used. Across all iteration limits and programs, DFO and Random orders each achieve better results than the other almost half of the time, but there are only two statistically significant differences between the two orders. These occur on `printtok1` and `replace` at the third iteration level, with DFO exhibiting better results once and Random once.

Where cost results for the genetic algorithm are concerned we see different trends. First, in the GDO vs GRO column there are 11 cases where order causes statistically significant differences: these include all results for `tcas` and `printtok1`. In the GDN vs GRN column there are also 11 cases, again including all cases for `tcas` and `printtok1`. In all but one of these cases, Random is more costly than DFO.

Turning to the concolic approach, where coverage is concerned, we do see an increase in the number of statistically significant differences between techniques, to 15 cases. However, in this case there is no clear superiority adhering to either of the two test case orders; each of Random and DFO are superior in several cases, and there are no apparent patterns involving iteration limits or programs to indicate factors potentially influencing this.

Finally, considering cost results for concolic, we again see a large number of statistically significant differences in costs, with 12 in the CDO vs CRO case and 14 in the CDN vs CRN case. Here, however, there is no clear advantage adhering to either Random or DFO orders: each is superior a number of times.

5.2 RQ2: Use of Existing and New Test Cases

Our second research question pertains to the effects of reusing existing and newly generated test cases. Table 8 presents data relevant to this question. The table format is similar to that of Table 7,

Table 3: Coverage Using DFO Order and Old Test Cases

	Coverage					Cost				
Genetic	*5*	*10*	*15*	*20*	*25*	*5*	*10*	*15*	*20*	*25*
`printtok1`	158.04	158.55	158.94	159.22	158.90	51.96	111.36	180.21	238.75	312.82
`printtok2`	176.97	177.01	177.08	177.09	177.09	36.39	77.13	118.99	166.07	224.83
`replace`	186.10	187.27	187.48	187.87	188.09	77.96	157.56	237.00	315.19	387.36
`tcas`	70.72	70.92	70.95	70.89	70.95	3.32	6.46	9.24	12.74	16.14
Concolic	*1*	*3*	*5*	*7*	*9*	*1*	*3*	*5*	*7*	*9*
`printtok1`	150.26	155.05	155.83	156.39	156.52	1.16	2.72	4.20	5.61	7.08
`printtok2`	168.81	172.86	173.29	173.78	174.38	0.19	0.39	0.56	0.76	0.91
`replace`	180.52	187.58	189.42	189.93	190.17	1.10	3.42	5.85	8.19	10.62
`tcas`	65.92	67.32	69.03	70.07	70.13	0.06	0.12	0.17	0.23	0.28

Table 4: Coverage Using DFO Order and Old and New Test Cases

	Coverage					Cost				
Genetic	*5*	*10*	*15*	*20*	*25*	*5*	*10*	*15*	*20*	*25*
`printtok1`	158.43	158.88	159.10	159.43	159.55	110.63	217.37	336.49	447.89	565.37
`printtok2`	177.07	177.13	177.09	177.15	177.17	65.66	130.14	189.58	279.77	348.29
`replace`	187.46	188.18	188.65	188.80	188.81	191.79	380.03	546.47	726.10	946.33
`tcas`	70.79	70.96	70.95	70.99	70.98	4.75	8.59	12.94	17.08	21.02
Concolic	*1*	*3*	*5*	*7*	*9*	*1*	*3*	*5*	*7*	*9*
`printtok1`	150.44	155.27	156.17	156.65	156.81	1.35	3.43	5.45	7.31	9.27
`printtok2`	169.00	173.14	173.60	174.12	174.77	0.21	0.43	0.64	0.86	1.04
`replace`	180.69	188.41	189.98	190.51	190.70	1.23	4.21	7.25	10.10	12.88
`tcas`	66.05	67.78	69.74	70.82	70.88	0.06	0.13	0.19	0.26	0.33

Table 5: Coverage Using Random Order and Old Test Cases

	Coverage					Cost				
Genetic	*5*	*10*	*15*	*20*	*25*	*5*	*10*	*15*	*20*	*25*
`printtok1`	158.15	158.40	158.59	158.93	159.18	69.40	147.63	223.41	297.22	395.00
`printtok2`	176.91	177.00	177.10	177.10	177.04	39.05	81.08	122.22	161.15	218.60
`replace`	186.22	187.37	187.92	188.12	188.22	79.09	148.94	219.75	303.07	385.72
`tcas`	70.61	70.82	70.95	70.96	70.98	3.84	7.04	10.63	14.76	18.78
Concolic	*1*	*3*	*5*	*7*	*9*	*1*	*3*	*5*	*7*	*9*
`printtok1`	150.23	155.01	155.75	156.16	156.25	1.18	2.80	4.27	5.52	6.86
`printtok2`	169.02	173.06	173.52	173.95	174.47	0.23	0.42	0.58	0.74	0.91
`replace`	180.52	187.58	189.42	189.92	190.18	1.06	3.43	5.88	8.19	11.07
`tcas`	66.32	67.43	68.92	70.01	70.12	0.07	0.12	0.16	0.17	0.22

Table 6: Coverage Using Random Order and Old and New Test Cases

	Coverage					Cost				
Genetic	*5*	*10*	*15*	*20*	*25*	*5*	*10*	*15*	*20*	*25*
`printtok1`	158.41	158.95	159.04	159.18	159.66	132.93	253.35	386.91	520.15	665.82
`printtok2`	177.10	177.18	177.14	177.11	177.11	66.01	123.12	193.22	251.84	324.07
`replace`	187.58	188.41	188.40	188.81	188.89	171.97	357.44	488.62	686.36	832.12
`tcas`	70.70	70.95	70.97	70.96	70.97	6.07	11.39	16.24	22.45	29.62
Concolic	*1*	*3*	*5*	*7*	*9*	*1*	*3*	*5*	*7*	*9*
`printtok1`	150.41	155.23	156.03	156.39	156.52	1.36	3.52	5.47	7.11	8.94
`printtok2`	169.24	173.25	173.75	174.20	174.77	0.23	0.48	0.65	0.84	1.04
`replace`	180.70	188.42	190.00	190.56	190.76	1.22	4.28	7.35	10.29	13.57
`tcas`	66.45	67.52	69.64	70.79	70.88	0.07	0.13	0.18	0.21	0.26

Table 7: Impact of Order in which Affected Elements are Considered on Coverage and Cost.

	Coverage				Cost			
	GDO vs GRO	GDN vs GRN	CDO vs CRO	CDN vs CRN	GDO vs GRO	GDN vs GRN	CDO vs CRO	CDN vs CRN
`printtok1`	R D ***D*** D R	D R D D R	D D D ***D D***	D D D ***D D***	***R R R R R***	***R R R R R***	R ***R*** R D D	R ***R*** R ***D D***
`printtok2`	D D R R D	R R R D D	***R R R R*** R	***R R R*** R =	***R*** R R D D	R D R ***D*** D	***R R R*** D D	***R R*** R D D
`replace`	R R ***R*** R R	R R D R R	= = = D R	R R R R R	R D D D D	D D D D D	***D R R*** R ***R***	***D R R R R***
`tcas`	D D = R R	D D R D D	***R*** R ***D*** D D	***R*** D ***D*** D =	***R R R R R***	***R R R R R***	***R*** R ***D D D***	***R*** R ***D D D***

Table 8: Impact of Reuse of Existing Test Cases on Coverage and Cost.

	Coverage				Cost			
	GDO vs GDN	GRO vs GRN	CDO vs CDN	CRO vs CRN	GDO vs GDN	GRO vs GRN	CDO vs CDN	CRO vs CRN
`printtok1`	***N*** N N N N	N ***N N*** N ***N***	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***
`printtok2`	N ***N*** N N N	N ***N*** N N N	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***	N ***N N N N***
`replace`	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***
`tcas`	N N = ***N*** N	N ***N*** N = O	N ***N N N N***	N N ***N N N***	***N N N N N***	***N N N N N***	***N N N N N***	***N N N N N***

but in keeping with the goal of comparing across test case reuse approaches the differences in terms compared all involve reuse approaches (Old versus New+old).

We begin by considering the results for the genetic algorithm. Where coverage is concerned, in all but three cases, the use of new test cases is superior to reusing only old test cases. There are also many cases where test case reuse approach has a statistically significant effect. These include eight comparisons in the case where DFO is used, and ten in the case where random order is used. Results vary across programs with `replace` exhibiting significant differences in all cases.

Where cost results for the genetic algorithm are concerned we see much larger effects: in all cases, the use of new test cases adds to costs, and the effect of doing so is statistically significant.

Turning to the concolic approach where coverage is concerned, here we see even stronger evidence that test case reuse approach matters, with the use of new test cases always more effective, and in all but three cases statistically significantly so.

Finally, considering cost results for concolic, we note significant differences in all but one case, again with greater costs adhering to the use of new test cases.

5.3 RQ3: Test Case Generation Algorithm

Our third research question pertains to the effects of using different test case generation algorithms, and we begin by comparing effectiveness. One issue to consider in doing this involves inherent differences in the test case generation *algorithms*. In Section 4.3 we described the reasoning behind using several iteration limits for each algorithm: we expect concolic and genetic algorithms to respond differently over different limits, and using different limits lets us observe techniques independent of the threat to internal validity that would attend the use of a single iteration limit.

Where comparisons of techniques are concerned, there is no inherent relationship between a given iteration limit for concolic and a given iteration limit for genetic; that is, concolic limits 1, 3, 5, 7, and 9 do not "correspond" in any way to genetic limits 5, 10, 15, 20, and 25. It follows that we cannot validly compare algorithms to each other on a per-iteration-limit basis. Instead, for each object program P, we locate the iteration limit L_g at which the genetic algorithm operates most effectively on P, and the iteration limit L_c at which the concolic algorithm operates best on P, and we compare the algorithms at their respective optimal iteration limits.

Table 9 presents data relevant to RQ3 with respect to algorithm effectiveness following the analysis procedure just described. The table provides data for each object program and for each of the four combinations of affected element ordering and test reuse strategies studied. An individual table entry indicates which technique achieved higher coverage, and italics indicate cases where the difference was statistically significant.

As the table shows, on every program but `replace`, the genetic algorithm outperforms the concolic algorithm, in each category in which they were compared. On `replace` the advantage goes to concolic. All differences were statistically significant.

Turning to efficiency, note that this comparison is complicated by the inherent differences in our two implementations. In fact, it is quite difficult to fairly compare our two implementations for efficiency because they are derived from different sources, they cannot be said to represent "optimal" implementations of the two algorithms. Thus we restrict ourselves to observing efficiency differences in a qualitative fashion. As data presented in Tables 3-6 shows, costs for the genetic algorithm range from times in the tens of seconds to times above 500 seconds, while costs for the concolic algorithm range from times in the tenths of seconds to times near 10 seconds. With our current implementations this represents a very large difference in favor of the concolic approach.

Table 9: Comparison of Coverage: Genetic vs Concolic

Program Program	GDO vs CDO	GDN vs CDN	GRO vs CRO	GRN vs CRN
`printtok1`	*G*	*G*	*G*	*G*
`printtok2`	*G*	*G*	*G*	*G*
`replace`	*C*	*C*	*C*	*C*
`tcas`	*G*	*G*	*G*	*G*

A further issue involves the effects that increasing iteration limits have on the respective algorithms. Here, as remarked earlier, increases in limits seem to correspond to roughly similar increases, proportionally, in costs. This provides some post-hoc justification for our choice of particular iteration limits, in that they seem somewhat comparable in terms of their effects on relative effort.

6. DISCUSSION AND IMPLICATIONS

We now discuss the results presented in the prior section, and comment on their implications for research and practice.

Test Case Order

Order of affected elements is not likely to significantly affect algorithm effectiveness because the same elements will ultimately be considered under any order, and this is what we saw in our study.

Where costs are concerned, in contrast, we do see some differences. Our results show that DFO can provide savings in costs when using the genetic algorithm. This can be explained by observing that with the genetic algorithm, if we work with higher-level branches first we can incidentally cover additional branches. Also, test cases that cover branches higher in dependency chains will have inputs that are close to those used to reach lower branches, thereby seeding the population with inputs that help the algorithm cover those more quickly.

With the concolic algorithm, in contrast, cost saving results are mixed. We suspect this is because test cases generated to cover b_t (lines 11-19 of Algorithm 3) may not cover other uncovered branches unless these uncovered branches share a common ancestor branch in a short distance from b_t (less than n_{iter}) in an execution tree. In such cases, the ordering of affected elements does not matter in terms of cost.

All things considered, we could argue that DFO has the potential to be more efficient than random ordering when using genetic algorithms, but the fact that this result occurs only for `printtok1` and `tcas` leads us to be cautious about this. Furthermore, there seems to be no clear benefit to using either order where the concolic approach is concerned. Still, these results do not preclude finding some other orderings that are more predictably cost-effective.

Test Case Reuse Approach

Our results show that the use of new test cases in addition to existing test cases almost always significantly increases the cost of test generation by both techniques. This result can be explained by the correlation between technique effort and the number of test cases used to seed the technique. Having additional test cases impacts both techniques: it controls the population size in the genetic algorithm, while the concolic technique must consider each test case supplied to it.

The use of new test cases also significantly increased test generation technique effectiveness in almost all cases in which the concolic approach was used, and in many cases where the genetic approach was used. This difference in results can be explained as follows. With the genetic algorithm, having additional test cases to work with can increase population diversity and improve the chances that crossover will generate chromosomes that cover previously uncovered branches; however, changes due to the increase might not be substantial when just a few test cases are added to those that had been used previously. The concolic approach, in contrast, utilizes each new test case independently and can gain from each as such.

If these results generalize we have a true cost-benefit tradeoff. With both techniques there is a potential payoff for incurring the additional costs involved in reusing test cases, and this effect is much larger for the concolic technique than for the genetic technique. Whether any effectiveness gain is worth the additional cost, however, must be assessed relative to the system being verified.

Test Case Generation Techniques

Concolic and genetic test case generation techniques did perform statistically significantly differently in our study, with the genetic algorithm exhibiting greater effectiveness than the concolic algorithm on `printtok1`, `printtok2`, and `tcas`, under all combinations of other factors. It appears that the genetic algorithm is more costly (potentially by two orders of magnitude) than the concolic algorithm in doing this, although again this comparison is complicated by the presence of several potentially confounding factors. These observations prompt us to explore possible causes for differences.

Generally speaking, concolic testing can generate test cases effectively as long as a target program does not contain many complex symbolic expressions, or utilize pointer arithmetic, non-linear arithmetic, and external library calls on symbolic variables, etc. This is because new test cases can be generated by the concolic approach only if a generated path condition can be solved by the underlying constraint solver.

Genetic algorithms may be more flexible than concolic, in that the chromosome and fitness can be adapted to many input types and data structures. The quality of the test cases generated and the algorithm cost, however, will be dependent on how well fitness is defined, and how well the parameters of the algorithm are tuned, and these will be application specific. For instance, if we set our mutation rate too high, or if our crossover, selection or fitness are not carefully designed, then we may fail to converge quickly causing longer run times. Similarly, since we must run all of our test cases to calculate fitness, if we use a population that is too large, this will negatively impact cost.

In the case of our study, our object programs do not contain complex symbolic expressions, but all of the programs except `tcas` take strings as inputs. Faced with string inputs, genetic algorithms can easily mutate these, covering additional branches, and they can attempt to use quite a few different mutants. Concolic algorithms cannot as easily address these programs, because they transform test cases only locally; that is, given a target branch b_t and a base path condition pc (line 10 of Algorithm 3), the concolic algorithm transforms a number of branches no greater than n_{iter}.

The inherent differences between concolic and genetic algorithms, and the observed differences in our study, suggest that augmentation techniques which combine both approaches, either using both on a particular target, or differentially applying one or the other depending on characteristics of a target, might be more cost-effective than approaches that utilize just single techniques.

Iteration Limits

We did not consider iteration limit to be an independent variable; rather, we blocked our analyses per iteration limit value, since this is our stopping criterion. We did examine our data, however, to assess iteration limit effects.

First, there does appear to be an increasing trend in coverage values as iteration limits increase. Beginning with the genetic algorithm, and considering the 16 cases in which limits increase (i.e., four increases per program, progressing from 5 to 10, 10 to 15, 15 to 20, and 20 to 25) coverage values for GDO increase as limits increase in 13 of 16 cases, coverage values for GRO increase as limits increase in 14 of 16 cases, coverage values for GDN increase as limits increase in 13 of 16 cases, and coverage values for GRN increase as limits increase in 11 of 16 cases. The coverage increases, however, are small overall — never more than two — and only eight are statistically significant, which indicates that our genetic algorithm is converging.

Iteration trends occur for the concolic algorithm as well, with values generally increasing by small amounts in all 64 cases. In this case, 63 of these increases are statistically significant, suggesting that iteration plays a more measurable role for the concolic approach than for the genetic approach, and that further increases may provide opportunities to increase effectiveness.

Where algorithm costs are concerned iteration limits have larger effects. For the genetic algorithm, costs differ across iteration limits by relatively substantial amounts (i.e., by factors ranging from four to six from iteration limits 5 to 25). Where the concolic algorithm is concerned we also see increases in costs as iteration limits increase. The increases are smaller numerically than those observed with the genetic algorithm, but they are similar in terms of the factors involved (i.e., they increase by factors ranging from three to ten from iteration limits 1 to 9).

7. CONCLUSIONS AND FUTURE WORK

In this work we have focused on test suite augmentation, and our results have several implications for the creation and further study of augmentation techniques. The results also have implications, however, for engineers creating initial test suites for programs. This is because such engineers often begin, at least at the system test level, with black box requirements-based test cases. It has long been recommended that such test suites be extended to provide some level of coverage. The techniques we have presented can conceivably serve in this context too, working with initial black-box test cases and augmenting these.

There are additional factors that influence augmentation that we have not examined directly in this work. Program characteristics certainly play a role, because they can impact the ability of test case generation techniques to function cost-effectively, as described in Sections 3.2 and 3.3. Characteristics of existing test suites also matter. Arguably, larger test suites, or test suites that are more comprehensive in the inputs that they provide or the coverage that they achieve, might be more cost-effective to augment. We attempted to control for such characteristics in our experiment by using initial test suites with varying sizes and coverage characteristics, but a more formal study of this factor could be helpful.

Acknowledgments

This work was supported in part by NSF under Awards CNS-0454203 and CCF-0747009 and by the AFOSR through award FA9550-09-1-0129, to the University of Nebraska - Lincoln. Also, this work was supported in part by the ERC of Excellence Program of Korea MEST/NRF (Grant 2010-0001727), the ITRC support program supervised by NIPA (NIPA-2010-(C1090-1031-0001)), and Basic Science Research Program through the Korea NRF funded by the MEST (2010-0005498). We thank Yuyang Liu for her valuable input on concolic testing of the object programs.

8. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 1986.

[2] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Test.: Acad. Ind. Conf. Pract. Res. Techn.*, pages 137–146, Aug. 2006.

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proc. Int'l Symp. Softw. Test. and Anal.*, July 2008.

[4] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *Proc. Int'l. Symp. Softw. Test. Anal.*, pages 108–118, July 2004.

[5] D. Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8), Aug. 1997.

[6] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proc. Conf. Comp. Comm. Sec.*, pages 322–335, Oct 2006.

[8] T. Y. Chen and R. Merkel. Quasi-random testing. *IEEE Trans. Rel.*, 56(3):562–568, 2007.

[9] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, Sept. 1976.

[10] L. Clarke and D. Richardson. Applications of symbolic evaluation. *J. Sys. Softw.*, 5(1):15–35, Jan. 1985.

[11] CREST - automatic test generation tool for C. `http://code.google.com/p/crest/`.

[12] R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *IEEE TSE*, 17(9):900–910, Sept. 1991.

[13] E. Díaz, J. Tuya, R. Blanco, and J. Javier Dolado. A tabu search algorithm for structural software testing. *Comp. Op. Res.*, 35(10):3052–3072, 2008.

[14] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Emp. Softw. Eng.: Int'l J.*, 10(4):405–435, 2005.

[15] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002.

[16] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proc. Int'l Symp. Softw. Test. Anal.*, pages 151–162, July 2007.

[17] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Meth.*, 5(1):63–86, Jan. 1996.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. Conf. Prog. Lang. Des. Impl.*, pages 213–223, June 2005.

[19] A. Gotlieb, B. Botella, and M. Reuher. Automatic test data generation using constraint solving techniques. In *Proc. Int'l. Symp. Softw. Test. Anal.*, pages 53–62, Mar. 1998.

[20] R. Gupta, M. Harrold, and M. Soffa. Program slicing-based regression testing techniques. *J. Softw. Test., Verif., Rel.*, 6(2):83–111, June 1996.

[21] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. Int'l. Conf. Softw. Eng.*, pages 191–200, May 1994.

[22] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–897, Aug. 1990.

[23] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Eng.*, 33(4):225–237, Apr. 2007.

[24] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. Int'l. Conf. Auto. Softw. Eng.*, Nov. 2001.

[25] P. McMinn. Search-based software test data generation: A survey. *J. Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[26] C. Michael, G. McGraw, and M. Shatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27(12):1085–1110, Dec. 2001.

[27] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proc. Int'l. Conf. UML*, Oct. 1999.

[28] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proc. Int'l. Symp. Found. Softw. Eng.*, Nov. 2004.

[29] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. Int'l. Symp. Found. Softw. Eng.*, pages 226–237, Nov. 2008.

[30] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proc. Int'l Symp. Softw. Test. Anal.*, Aug 1994.

[31] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Meth.*, 6(2):173–210, Apr. 1997.

[32] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. Int'l Conf. Auto. Softw. Eng.*, Sept. 2008.

[33] K. Sen and G. Agha. JCUTE: Concolic unit testing and explicit path model-checking tools. In *Proc. Int'l Conf. Comp. Aided Verif.*, pages 419–423, Aug 2006.

[34] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. Int'l Symp. Found. Softw. Eng.*, pages 263–272, Sept. 2005.

[35] P. Tonella. Evolutionary testing of classes. In *Intl. Symp. Softw. Test. Anal.*, pages 119–128, 2004.

[36] W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with Java Pathfinder. In *Proc. Int'l Symp. Softw. Test. Anal.*, pages 97–107, July 2004.

[37] H. Waeselynck, P. Thévenod-Fosse, and O. Abdellatif-Kaddour. Simulated annealing applied to test generation: Landscape characterization and stopping criteria. *Emp. Softw. Eng.*, 12(1):35–63, 2007.

[38] A. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *Proc. Int'l. Conf. Softw. Test. Anal.*, pages 1–12, July 2006.

[39] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Conf. Gen. Evol. Comp.*, pages 1053–1060, 2005.

[40] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proc. Int'l Symp. Softw. Test. Anal.*, pages 249–260, July 2008.

[41] Z. Xu, M. Cohen, and G. Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *Gen. Evol. Comp. Conf*, July 2010.

[42] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Proc. Asia-Pacific Softw. Eng. Conf.*, Dec. 2009.