2011 22nd IEEE International Symposium on Software Reliability Engineering

A Hybrid Directed Test Suite Augmentation Technique

Zhihong Xu
Computer Science and Engineering
U. of Nebraska - Lincoln
zxu@cse.unl.edu

Yunho Kim, Moonzoo Kim
Computer Science
KAIST
kimyunho@kaist.ac.kr
moonzoo@cs.kaist.ac.kr

Gregg Rothermel
Computer Science and Engineering
U. of Nebraska - Lincoln
grother@cse.unl.edu

***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. In previous work, we studied two approaches to augmentation, one using a concolic test case generation algorithm and one using a genetic test case generation algorithm. We found that these two approaches behaved quite differently in terms of their costs and their abilities to generate effective test cases for evolving programs. In this paper, we present a hybrid test suite augmentation technique that combines these two test case generation algorithms. We report the results of an empirical study that shows that this hybrid technique can be effective, but with varying degrees of costs, and we analyze our results further to provide suggestions for reducing costs.**

I. 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 new system versions. *Test suite augmentation techniques* (e.g., [1], [23], [27], [35]) address this problem, by identifying where new test cases are needed and then generating them.

In prior work, we have created and investigated several different approaches to test suite augmentation, focusing on the use of test case generation techniques that take advantage of existing test suites. Our initial *directed test suite augmentation technique* uses a regression test selection algorithm [26] 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 algorithm [28] to create test cases that execute the affected code. A case study showed that the approach can improve both the efficiency of the technique and its ability to cover affected code. Further work [33] examined a similar approach to augmentation using a genetic algorithm for test case generation, also with promising results.

More recently, we empirically studied several factors that can affect augmentation techniques [34]. These include (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. The results of our studies show that the primary factor affecting augmentation is the test case generation algorithm utilized; this affects both the cost and the effectiveness of augmentation techniques. The manner in which existing and newly generated test cases are utilized also has a substantial effect on technique efficiency, and when using a concolic test case generation algorithm can have a substantial effect on effectiveness. The order in which affected elements are considered turns out to have relatively few effects when using a concolic test case generation algorithm, but in some cases can influence the efficiency of genetic test case algorithms.

The inherent differences between concolic and genetic test case generation algorithms and the observed differences in the foregoing studies lead us to conjecture that augmentation techniques that combine both algorithms should be more cost-effective than approaches that utilize just a single algorithm. Such *hybrid* test suite augmentation techniques could provide an overall augmentation approach that addresses the algorithmic limitations seen in individual techniques.

In this work we investigate this possibility further. We present an approach for combining test case generation techniques into a *hybrid directed test suite augmentation technique*. We present results obtained by applying our hybrid technique to a set of C programs, comparing its cost and effectiveness to the use of basic augmentation approaches that apply only concolic or genetic test case generation algorithms. Our results show that hybrid techniques can be more effective than non-hybrid techniques; however, the results also showed (contrary to our expectations) that hybrid techiques do not necessarily yield greater efficiency than non-hybrid techniques. Given this latter result, we further analyze our data and provide suggestions on ways to create more cost-effective hybrid algorithms.

II. Background and Related Work

A. 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 (e.g., [21], [26]) and test case prioritization (e.g., [12], [18], [31]).

1071-9458/11 $26.00 © 2011 IEEE
DOI 10.1109/ISSRE.2011.21

IEEE computer society

Test suite augmentation techniques, in contrast, are not concerned directly 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 [4] 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. Early code level techniques [3], [14], [25] 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.

Appiwattanapong et al. [1] and Santelices et al. [27] 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. Person et al. (2008) [23] present an approach to program differencing using symbolic execution that can be used to identify the effects of program changes, and yields constraints that can be input to a solver to generate test cases for those requirements. Person et al. (2011) [24] use program analysis techniques to identify the parts of new programs that are affected by changes and apply symbolic execution only on these parts. None of the foregoing approaches, however, are integrated with reuse of existing test cases.

Only a few papers have considered augmentation from the standpoint of reusing and generating new test cases. We have already described, in Section I, our own work in this area [33], [34], [35], in which adaptations of genetic and concolic test generation techniques use test resources and data obtained from prior testing sessions to generate test cases to cover target code elements. More recently, Xie et al. [29] presented an approach for using dynamic symbolic execution to reveal execution paths that need to be retested, in which existing test cases can be utilized.

B. Automated 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 (e.g., [6]) from formal models (e.g., [15]) and by random selection of inputs (e.g., [7]). Several other techniques (e.g., [8]) use symbolic execution to find the constraints, in terms of input variables, that must be satisfied to execute a target path, and attempt to solve this system of constraints to obtain a test case for that path.

More recently, test case generation techniques that rely on dynamic information have appeared. Several such techniques use evolutionary or search-based approaches (e.g., [2], [10], [22]) such as genetic algorithms, tabu search, and simulated annealing to generate test cases. Other work (e.g., [5], [13], [28]) 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. In our work, we focus on these two classes of approaches, because they can make use of existing test cases, and because such test cases are readily available in a regression testing scenario. Here, we summarize these overall approaches; details on our adaptations of them are presented in Section III.

1) Genetic Test Case Generation: Genetic algorithms for structural test case generation begin with an initial (often randomly generated) test case population and evolve the population toward targets that can be blocks, branches or paths in a program [20], [30], [32]. 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.

2) Concolic Test Case Generation: Concolic testing [5], [13], [28] 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 [28]. 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

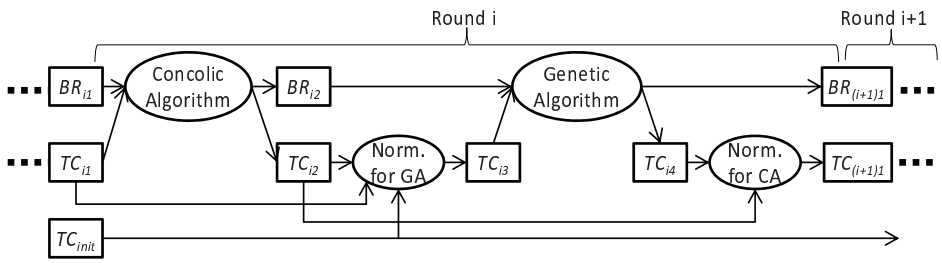

Figure 1. Overview of hybrid test suite augmentation approach

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 practice, bounds on path length or algorithm run-time can be applied).

3) Combination of techniques: Recently, other researchers have combined different techniques to help generate test cases. Hybrid concolic testing [19] combines random and concolic testing to generate test cases. In contrast, our technique combines genetic and concolic techniques, and we focus on the test suite augmentation context, in which there are many other factors to be considered that are not discussed in [19]. Inkumsah et al. [17] combine a genetic algorithm and concolic testing to generate test cases for programs. They focus on unit testing of objected-oriented programs, whereas we focus on on system testing. Further, they use evolutionary testing to find method sequences and concolic testing to cover branches, whereas our hybrid approach uses the two generation methods together to enhance branch coverage. Finally, their approach does not reuse existing test cases, which is central to our approach.

III. Hybrid Test Suite Augmentation

The results of our most recent study of test suite augmentation techniques [34], summarized in Section I, suggest that a hybrid test suite augmentation technique should be created keeping the following requirements in mind:

1) Concolic test case augmentation is much more efficient than genetic test case augmentation. Thus, a hybrid technique should begin by using a concolic test case generation algorithm and attempt to cover as many branches as possible before passing control to a genetic test case generation algorithm.
2) Processing targets in depth-first order can improve the efficiency of the genetic algorithm but has no effect on the concolic algorithm. Thus, we can order the targets to improve the former without harming the latter.
3) Test reuse approach has an impact on the effectiveness of the concolic algorithm. When using that algorithm we should utilize new test cases as they are created.

Our hybrid test suite augmentation technique is summarized in Figure 1. This hybrid technique incorporates multiple *rounds* of test case generation, where one round consists of an application of a concolic test case generation algorithm followed by an application of a genetic test case generation algorithm. We focus on branch coverage rather than path coverage for issues of scalability; rounds continue until no new branches are covered. In the ith round, the concolic algorithm receives a list of target branches BR_{i1} and a set of test cases TC_{i1} from the (i-1)th round, where BR_{11} is a list of all target branches sorted in depth-first order and $TC_{11} = TC_{init}$ is a set of initial test cases.[1] For each round i:

1) The concolic algorithm generates a set of new test cases TC_{i2}, each of which covers at least one new branch (see Section III-A for details). After this step, $BR_{i2} = BR_{i1} - cov(TC_{i2})$, where $cov(TC)$ indicates a set of branches covered by TC_{i2}.
2) TC_{init}, TC_{i1} and TC_{i2} are *normalized/modified* to form a test case population TC_{i3} for genetic testing. Currently, the genetic algorithm employed by our hybrid augmentation technique fixes the size of a test case population at $|TC_{init}|$ for all rounds (i.e.,$\forall i \geq 1, |TC_{i3}| = |TC_{init}|$). This normalization process randomly selects $|TC_{init}|$ test cases from $TC_{init} \cup TC_{i1} \cup TC_{i2}$.
3) The genetic algorithm generates a set of test cases TC_{i4}, each of which covers at least one new branch (see Section III-B for details). After this step, $BR_{(i+1)1} = BR_{i2} - cov(TC_{i4})$.
4) TC_{i2} and TC_{i4} are normalized to form $TC_{(i+1)1}$, a set of test cases that is used by the concolic algorithm in the $(i+1)$th round. Currently, this step sets $TC_{(i+1)1}$ to $TC_{i2} \cup TC_{i4}$, which are new test cases. This step enables the concolic algorithm to utilize the "old+new test case reuse strategy" (requirement 3).

[1] In the first round, any set of branches determined to need coverage can be passed to the algorithm; in this work we assume that a regression test suite has been executed on the program, and that the initial set of branches is the set of branches not covered by the test cases in that suite.

Input: a set of test cases TC, a set of target branches BR, an uncovered target branch $b_t \in BR$, and an iteration limit n_{iter}
Output: a set of new test cases NTC and a remaining set of target branches NBR

1 $NTC = \emptyset$ // new test cases
2 $NBR = BR$ // new target branches
3 $TC_{\overline{b_t}} = \{$ all test cases in TC that reach $\overline{b_t}$ $\}$
4 **if** $TC_{\overline{b_t}} = \emptyset$ **then**
5 | **return** $\emptyset$ *and* NBR
6 **end**
7 $PC_{\overline{b_t}} = \{$path conditions from executing tests in $TC_{\overline{b_t}}\}$
8 **foreach** $pc \in PC_{\overline{b_t}}$ **do**
9 | **foreach** $i = LastPos(\overline{b_t}, pc)$ **to** $i - n_{iter}$+1 **do**
10 | | **if** $i > 0$ **then**
11 | | | $pc' = DelNeg(pc, i)$
12 | | | $tc_{new} = Solve(pc')$
13 | | | **if** $tc_{new} \neq$ UNSAT and tc_{new} covers uncovered branches in NBR **then**
14 | | | | $NBR = NBR-$ newly covered branches by tc_{new}
15 | | | | $NTC = NTC \cup \{tc_{new}\}$
16 | | | **end**
17 | | **end**
18 | **end**
19 **end**
20 **return** NTC *and* NBR

Algorithm 1: CONCOLIC−AUGMENT algorithm

Input: a set of test cases TC, a set of target branches BR, an uncovered target branch $b_t \in BR$, and an iteration limit n_{iter}
Output: a set of new test cases NTC and a remaining set of target branches NBR

1 $TC_{cur} = TC$ // current target test cases
2 $NTC = \emptyset$ // new test cases
3 $NBR = BR$ // new target branches
4 $TC_{b_t} = \{$test cases in TC_{cur} that reach method m_{b_t}, the method containing $b_t\}$
5 $Population = TC_{b_t}$
6 $i = 0$
7 **repeat**
8 | $Fitness$=CalculateFitness($Population$)
9 | $Population$=Select($Population$, $Fitness$)
10 | $Population$=Crossover($Population$)
11 | $Population$=Mutate($Population$)
12 | $i = i + 1$
13 | **foreach** $tc \in Population$ **do**
14 | | Execute (tc)
15 | | **if** tc covers new branches in NBR **then**
16 | | | $NBR = NBR-$ newly covered branches
17 | | | $NTC = NTC \cup \{$tc$\}$
18 | | **end**
19 | **end**
20 **until** $i \geq n_{iter}$ or b_t is covered;
21 **return** NTC *and* NBR

Algorithm 2: GENETIC-AUGMENT algorithm

The precise algorithms used for concolic and genetic test case generation in the foregoing hybrid augmentation technique are similar to those used in our earlier work [34]. For completeness we present those algorithms with a somewhat abbreviated discussion in the following sections.

A. Concolic Test Suite Augmentation

We use the following notations. A *path condition* pc for a target program is a conjunction $b_{i_1} \wedge b_{i_2} \wedge ...b_{i_n}$ where $b_{i_1}, ...b_{i_n}$ are branch conditions in the target program and executed in order. $DelNeg(pc, j)$ generates a new path condition from 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 pc where $b = b_{i_j}$ (i.e., $\forall j < k \leq n.b_{i_k} \neq b$).

Algorithm 1 repeats for each target branch $b_t \in BR$ that has not yet been covered. Initially, a set of new test cases NTC is empty (line 1) and a set of target branches to cover $NBR = BR$ (line 2). The start of the main procedure selects test cases that can reach $\overline{b_t}$ from among TC (line 3). If there are no such test cases, the algorithm terminates (lines 4-6). If there are such test cases, the algorithm obtains path conditions by executing the target program with selected test cases (line 7). From each obtained path condition pc (lines 8-19), 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 9-18) by negating $b_{i_m}, b_{i_{m-1}}, ..., b_{i_{m-n_{iter}+1}}$ and removing all following branches in pc, respectively (line 11).[2] If a newly generated path condition pc' has a solution tc_{new} (a new test case) (line 12) and tc_{new} covers uncovered branches in NBR (line 13), NBR is updated to reflect the new status of coverage (line 14), and tc_{new} is added to the set of newly generated test cases NTC (line 15).

B. Genetic Test Suite Augmentation

Algorithm 2 repeats for each target branch $b_t \in BR$ that has not yet been covered. Algorithm 2 iterates for a number of generations (set by variable n_{iter}) or until b_t is covered. The first step (line 8) is to calculate the fitness of all test cases in the test case population. Since the fitness of a test case depends on its relationship to the branch the algorithm is trying to cover, calculating the fitness requires that the test case be run. (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 9), which orders and chooses the best half of the chromosomes to use in the next step. The 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

[2] n_{iter} 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.

have been mated. Next (line 11) a small percentage of the population is mutated, after which all test cases in the current population are executed.

IV. Empirical Study

Our goal is to compare the use of our hybrid directed test suite augmentation technique to non-hybrid techniques. We thus pose the following research questions.

RQ1: How does hybrid test suite augmentation compare, in terms of cost and effectiveness, to augmentation using a straightforward concolic test case generation technique?

RQ2: How does hybrid test suite augmentation compare, in terms of cost and effectiveness, to augmentation using a straightforward genetic test case generation technique?

A. Objects of Analysis

To facilitate augmentation technique comparisons, programs must be suitable for use by all techniques. Also, programs must be provided with test suites that need to be augmented. In our prior work we had selected several programs from the SIR repository [11] that meet the needs of such comparisons. Here we utilize three of these programs (see Table I). Each program 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 [16].

Table I
Experiment Objects

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

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. In prior work, however, we defined 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 in which 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, and we applied this algorithm 1000 times to P. Next, we measured the minimum size T_{min} and maximum size T_{max} for these suites; this provides estimates of the lower and upper size bounds for coverage-adequate test suites for the programs. Because in practice, programs are often equipped with test suites that are not coverage-adequate, and because we wish to study the effects of augmentation using a wide range of initial test suite sizes and coverage characteristics, we set lower and upper bounds for initial test suites at $T_{min}/2$ and T_{max}, respectively.

Second, we began the test suite construction phase, in which for each test suite to be constructed, we randomly chose a number n such that $T_{min}/2 \leq n \leq T_{max}$, 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 II.

Table II
Branch Coverage and Sizes of Initial Test Suites

Program	*Branch Coverage*			*Test Suite Size*		
	Avg	Min	Max	Avg	Min	Max
`printtok1`	133.3	110.0	152.0	16.8	9	25
`printtok2`	158.8	129.0	173.0	18.4	8	29
`replace`	165.9	127.0	182.0	17.8	9	28

B. Variables and Measures

The comparison of hybrid and non-hybrid techniques is complicated by the fact that they inherently involve different amounts of effort. One could certainly run the two types of techniques for the same amount of time and compare their relative effectiveness, but we expect that in practice, engineers would run the techniques until the techniques cease to achieve sufficient new coverage, and then stop. It thus seems more appropriate to run the techniques to some reasonable stopping points, and then compare their relative effectiveness and efficiency. We choose independent and dependent variables keeping this approach in mind. Further, as discussed below, we use different iteration limits to investigate the variance that might be seen in performance if techniques are allowed to run longer times.

Independent Variable. Our experiment manipulates one independent variable: the augmentation technique used. Three treatments were chosen for this variable: (1) the hybrid test suite augmentation technique described in Section III, (2) an augmentation technique using just concolic test case generation, and (3) an augmentation technique using just genetic test case generation.

Dependent Variable. We wish to measure both the effectiveness and the efficiency 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 effec-

tiveness of techniques, we track the number of branches in P' that can be covered by each augmented test suite.

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

C. Experiment Setup and Operation

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

1) Extended Programs: Our concolic test suite augmentation technique is implemented based on CREST [9]. 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 techniques that use 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 three programs, and apply all techniques to those versions.

2) Iteration Limits: Genetic algorithms iteratively generate test cases, and an iteration limit governs the stopping point for this activity. Similarly, the concolic algorithm 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 efficiency of the algorithms. Moreover, as noted above, in practice engineers might select different stopping points for the algorithms. For these reasons, in our experiments we use multiple iteration limits for each test case generation algorithm, choosing 1-5-9 for concolic and 5-15-25 for genetic because exploratory studies showed that these represented lower and upper bounds outside of which technique effectiveness ceased to vary by more than small amounts.

3) Technique Settings and 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. In our genetic algorithm, we chose simple methods for both selection and crossover as mentioned in Section III-B. We used the approach level in the fitness function. Also for different programs, we used different mutation rates: 0.06 for `printtok1` and `printtok2` and 0.08 for `replace`.

4) Experiment Operation: 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.

D. 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 three relatively small C programs using simulated versions, and the study of other objects, other types of versions, and other test suites may exhibit different cost-benefit tradeoffs. However, if results on smaller programs show that our approach is beneficial, then arguably, programs with more complex features should enable a hybrid approach to function even better. 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 efficiency 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.

E. Results

Tables III, IV, and V present the data obtained in our study for the three object programs, respectively. Each table shows cost and coverage data. Data is shown per iteration limit, with CA1, CA5, and CA9 representing limits for the concolic test case generation algorithm, and GA5, GA15, and GA25 representing limits for the genetic test case generation algorithm. A given cell in the table represents a comparison between the techniques indicated by the label at the top of the column containing that cell.

Next we analyze our results, per research question.

1) RQ1: Hybrid versus concolic: The columns labeled "CA HY" in Tables III, IV, and V present data relevant to this question. Each entry in these columns shows the comparison between the hybrid test suite augmentation technique and the concolic test suite augmentation technique in terms of cost or coverage. The numbers represent the average cost of, or coverage obtained by, the two techniques across all 100 test suites. For example, the first entry in Table III

Table III
COVERAGE AND COST DATA FOR PRINTTOK1

COST (seconds)						
	CA1		CA5		CA9	
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***1.54 57.51***	56.38 57.51	***6.94 67.11***	***56.38 67.11***	***12.27 75.06***	***56.38 75.06***
GA15	***1.54 190.37***	***210.56 190.37***	***6.94 200.15***	***210.56 200.15***	***12.27 192.33***	***210.56 192.33***
GA25	***1.54 351.87***	339.19 351.87	***6.94 405.57***	***339.19 405.57***	***12.27 414.33***	***339.19 414.33***
COVERAGE (branches)						
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***143.97 155.58***	***154.89 155.58***	***151.29 155.65***	***154.89 155.65***	***152.50 155.78***	***154.89 155.78***
GA15	***143.97 156.11***	155.88 156.11	***151.29 156.23***	155.88 156.23	***152.50 156.02***	155.88 156.02
GA25	***143.97 156.62***	156.54 156.62	***151.29 156.51***	156.54 156.51	***152.50 156.51***	156.54 156.51

Table IV
COVERAGE AND COST DATA FOR PRINTTOK2

COST (seconds)						
	CA1		CA5		CA9	
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***0.25 35.67***	***32.21 35.67***	***0.84 35.02***	***32.21 35.02***	***1.43 31.86***	32.21 31.86
GA15	***0.25 153.88***	***131.25 153.88***	***0.84 154.71***	***131.25 154.71***	***1.43 159.42***	***131.25 159.42***
GA25	***0.25 275.49***	***248.64 275.49***	***0.84 291.97***	***248.64 291.97***	***1.43 296.92***	***248.64 296.92***
COVERAGE (branches)						
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***165.34 176.06***	***175.85 176.06***	***171.59 176.42***	***175.85 176.42***	***173.00 176.41***	***175.85 176.41***
GA15	***165.34 176.58***	***176.34 176.58***	***171.59 176.54***	***176.34 176.54***	***173.00 176.60***	***176.34 176.60***
GA25	***165.34 176.62***	***176.40 176.62***	***171.59 176.67***	***176.40 176.67***	***173.00 176.65***	***176.40 176.65***

Table V
COVERAGE AND COST DATA FOR REPLACE

COST (seconds)						
	CA1		CA5		CA9	
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***0.74 84.66***	***90.49 84.66***	***4.40 75.99***	***90.49 75.99***	***8.04 82.55***	***90.49 82.55***
GA15	***0.74 341.95***	***320.71 341.95***	***4.40 322.79***	320.71 322.79	***8.04 322.19***	320.71 322.19
GA25	***0.74 570.88***	***618.83 570.88***	***4.40 552.71***	***618.83 552.71***	***8.04 576.28***	***618.83 576.28***
COVERAGE (branches)						
	CA HY	GA HY	CA HY	GA HY	CA HY	GA HY
GA5	***176.43 186.94***	***185.80 186.94***	***187.24 190.02***	***185.80 190.02***	***188.59 190.53***	***185.80 190.53***
GA15	***176.43 188.58***	***187.83 188.58***	***187.24 190.51***	***187.83 190.51***	***188.59 190.75***	***187.83 190.75***
GA25	***176.43 189.18***	188.81 189.18	***187.24 190.66***	***188.81 190.66***	***188.59 190.88***	***188.81 190.88***

contains 1.54 and 57.51. Here, 1.54 represents the average cost in seconds to perform test suite augmentation across 100 test suites with the concolic augmentation technique run at iteration limit 1, while 57.51 represents the average cost in seconds when the hybrid augmentation technique is used with its concolic algorithm component run at iteration limit 1 and its genetic algorithm component run at iteration limit 5. For each pair of data sets (each cell in the tables), we applied a *Wilcoxon test* to determine whether there is a statistically significant difference between the two techniques, using $\alpha = 0.05$ as the confidence level. In the table, bold-italicized fonts indicate statistically significant differences. For example, for the first entry of Table III, comparing the costs of the hybrid augmentation technique and the concolic augmentation technique, there is a statistically significant difference between these two, and the concolic technique cost less than the hybrid technique.

We begin by considering comparisons in terms of cost. The concolic technique cost less than the hybrid technique on all programs, and the differences in cost were statistically significant in all cases. On `printtok1`, the hybrid technique cost up to 350 times more than the concolic technique; on `printtok2`, the hybrid technique cost up to 110 times more than the concolic technique; and on `replace`, the hybrid technique cost up to 771 times more than the concolic technique. (All of these maximal differences occurred when

the concolic technique was run at iteration limit 1 and the genetic component of the hybrid technique was run at iteration limit 25.)

Where effectiveness is concerned, the hybrid technique has advantages. In all entries related to coverage comparisons between the hybrid technique and the concolic technique, the hybrid technique covers more branches than the concolic technique, and the differences are statistically significant in all cases. On `printtok1`, the hybrid technique covered up to 13 branches more than the concolic technique; and on `printtok2` and `replace`, the hybrid technique covered up to almost 13 branches more than the concolic technique. Maximal differences occurred when the concolic technique was run at iteration limit 1 and the genetic component of the hybrid technique was run at iteration limit 25.

To summarize, comparing the concolic test case augmentation technique to the hybrid technique, the hybrid technique was more effective but less efficient.

2) RQ2: Hybrid versus genetic: The columns labeled "GA HY" in Tables III, IV, and V present data relevant to this question. We again begin with cost comparisons. Here, results varied more widely than in the case of RQ1. On `printtok1`, the hybrid augmentation technique cost more (by up to 33%) than the genetic augmentation technique in six of nine cases, of which four involve statistically significant differences. The genetic technique cost more (by up to 11%) than the hybrid technique in three cases, all of them statistically significant differences occurring when the genetic component of the hybrid technique was run at iteration limit 15. On `printtok2`, the hybrid technique cost more (by up to 21%) than the genetic technique in eight of nine cases, all of which involve statistically significant differences. The only exception occurred when the concolic component of the hybrid algorithm was run at iteration limit 9 and the genetic component was run at iteration limit 5, in which case the two did not differ significantly. On `replace`, the genetic technique cost more (by up to 19%) than the hybrid technique in six cases, all of which involved statistically significant differences. The genetic technique cost less in the other three cases, only one of which involved a statically significant difference.

In terms of coverage, on `printtok1` the hybrid technique achieved higher coverage than the genetic technique in seven cases, of which three involved statistically significant differences. The genetic technique had better coverage in the other two cases but with no statistically significant differences, and in both situations the differences were smaller than one branch. On `printtok2` and `replace`, the hybrid technique achieved higher coverage in all cases in which there are statistically significant differences. On `printtok2` the differences were less than one branch while on `replace`, the differences ranged from less than one branch up to almost five branches.

Overall, comparing the genetic test suite augmentation technique and the hybrid test suite augmentation technique, the hybrid technique achieved greater coverage than the genetic technique and sometimes (but not always) cost less.

V. Discussion and Implications

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

The hybrid test case augmentation technique outperformed both the concolic and genetic augmentation techniques in terms of effectiveness in most cases. If our results generalize, then when effectiveness has the highest priority, the hybrid technique is the best choice. In this respect, the results of our study met our expectations.

Where the cost of augmentation techniques is concerned, however, the results presented some surprises. On one hand, it is obvious that the hybrid technique should cost more than the concolic technique, because the hybrid technique includes a genetic algorithm component, which itself requires much more time than the concolic technique. On the other hand, we had expected the hybrid augmentation technique to cost less than the genetic augmentation technique, because the hybrid technique begins with a concolic test case generation step, which should cover some targets in a relatively short time, leaving fewer targets for the genetic algorithm to work on. We did observe this result in most cases on `replace`. On `printtok1` and `printtok2`, however, the hybrid technique usually did *not* save time with respect to the genetic technique. We inspected our results further and found that there are two reasons that can account for this difference.

A. Masked-out Benefit of Concolic Testing

The first reason for the performance difference is that the branches covered by the concolic algorithm component of the hybrid technique are easily covered by the genetic algorithm component of the hybrid technique, in the first few iterations of the genetic algorithm component. This means that the benefits of concolic testing (i.e., coverage of target branches in a relatively short time compared to the genetic algorithm) can be "masked out" at the beginning of the genetic algorithm. To further investigate this, we identified branches covered by the concolic algorithm and branches covered by the genetic algorithm in the first five iterations (note that in this case we applied both algorithms separately, not in the hybrid framework). Then, we calculated the percentage of branches that are covered by both algorithms over branches covered by the concolic algorithm. Table VI shows these percentage numbers. For example, the entry 53.26% in column CA1 for `printtok1` means that the straightforward genetic algorithm covers 53.26% of the branches in five iterations that are covered by the concolic algorithm with iteration limit 1. As the table shows, on `printtok1`, the genetic algorithm covers more than 53%

Table VI
BRANCHES COVERED BY BOTH ALGORITHMS OVER BRANCHED COVERED BY THE CONCOLIC ALGORITHM

	CA1	CA5	CA9
`printtok1`	53.26%	56.92%	55.15%
`printtok2`	79.49%	72.88%	69.52 %
`replace`	35.15%	32.83%	32.47%

Table VII
COST DIFFERENCES BETWEEN HYBRID ALGORITHMS

		printtok1			printtok2			replace		
		GA			GA			GA		
		5	15	25	5	15	25	5	15	25
CA	1	***H2***	***H2***	***H2***	H1	***H2***	***H2***	H1	***H2***	***H2***
	5	***H2***	H2	***H2***	H1	***H2***	***H2***	H1	***H2***	***H2***
	9	***H2***	H1	***H2***	H1	***H2***	***H2***	H1	***H2***	***H2***

of the branches covered by the concolic algorithm across all levels. On `printtok2`, the genetic algorithm covers even more branches: up to 79% of those covered by the concolic algorithm. Thus, this can explain why the hybrid algorithm is slower than the genetic algorithm on `printtok2`, since even more benefits of the concolic algorithm are masked out in this case. On `replace`, in contrast, the genetic algorithm covers fewer branches, so the benefits of using the concolic algorithm first are realized to a larger extent, and the hybrid technique saves time compared to the genetic technique.

B. Weakened Diversity of Test Case Population

The second reason for the performance difference involves the diversity of the test case population. In the hybrid technique we randomly select test cases from the existing test cases and the test cases newly generated by the concolic algorithm to form an initial population of test cases for use by the genetic algorithm. The test cases generated by the concolic algorithm, however, tend to be only slightly different from existing test cases, due to the manner in which the concolic algorithm operates. Thus, when drawing from these newly generated test cases it is more likely that an initial population of test cases will lack diversity, and this can reduce the efficiency of the genetic algorithm.

To further investigate this, we performed an additional set of runs using a version of the hybrid technique in which the genetic algorithm uses only test cases from the initial test suite TC_{init} to form the initial population for targets. When we compare the coverage data from these runs to the coverage data reported in Section IV-E, there are no statistically significant differences. When we compare the cost data from these runs, however, in most cases this new version of the hybrid algorithm (H2) outperformed the initial one (H1). Table VII shows the cost comparison between the two approaches. Table entries of "H1" indicate that the first hybrid algorithm cost less than the second one, while entries of "H2" indicate that the second algorithm cost less. Bold-italicized entires indicate that there is a statistically significant difference between the techniques. As the table shows, in most cases H2 cost significantly less than H1. This confirms our conjecture that the newly generated test cases affect the diversity of the population for the genetic algorithm, since this is the only differences between the two hybrid techniques. Nevertheless, H2 continues to have the shortcoming mentioned earlier (masked-out benefits of concolic testing) and does not significantly improve efficiency.

C. Potential Remedies

The foregoing discussion reveals several ways in which our basic hybrid algorithm could be improved. One method for overcoming the masked-out benefit of concolic testing (Section V-A) is to customize a concolic algorithm to attempt to reach branches that are difficult for a genetic algorithm to reach first. For example, it is well known that deeply nested branches are difficult for genetic algorithms to cover. We can modify a concolic algorithm to focus on such branches first. We can also modify the genetic algorithm to target branches that are difficult for the concolic algorithm to cover due to the presence of external libraries or floating point arithmetic.

Regarding the weakened diversity problem (Section V-B), we can select only new test cases generated by concolic testing that are largely different from each other as an initial population for genetic testing. Alternatively, we can enhance symbolic path formulas to generate a solution that is much different from the previous one by inserting additional constraints on the solution space. Last, we can fully utilize the randomized capability of an underlying SMT solver to obtain more diverse solutions.

VI. CONCLUSIONS AND FUTURE WORK

We have presented a hybrid technique for performing test suite augmentation, that utilizes both concolic and genetic test case generation algorithms in an attempt to harness the different strengths of both. Our empirical study of this technique shows that it can improve augmentation effectiveness, but as initially configured, it does not consistently save time in comparison to the genetic and concolic test suite augmentation techniques. Our analysis of these results uncovers reasons for this effect, and supports suggestions on how to improve the hybrid technique.

In this work we have focused on test suite augmentation. Our results also have implications, however, for engineers creating initial test suites for programs. Engineers often begin, at least at the system test level, with black box requirements-based test cases. The techniques we have presented can conceivably help these engineers extend initial black-box test cases to achieve better code coverage.

In future work we intend to improve our hybrid technique by following these suggestions, and study its application to additional and larger object programs. As further potential

improvements we will also seek ways in which the individual test case generation algorithms used by the hybrid technique can make use of additional information gathered by the other algorithms to generate test cases more cost-effectively.

Acknowledgements

This work was supported in part by the AFOSR through award FA9550-09-1-0129 to the University of Nebraska - Lincoln, through the Korea Research Foundation under award KRF-2007-357-D00215, and by the ERC of Excellence Program of Korea MEST/NRF (Grant 2011-0000978). Myra Cohen helped in earlier stages of the work on genetic test case augmentation.

References

[1] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *TAIC-PART*, pages 137–146, Aug. 2006.

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

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

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

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

[6] J. Chang and D. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Symp. Found. Softw. Eng.*, pages 285–302, Sept. 1999.

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

[8] L. Clarke. A system to generate test data and symbolically execute programs. *Trans. Softw. Eng.*, 2:215–222, May 1976.

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

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

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

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

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

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

[15] A. Hartman and K. Nagin. Model driven testing - agedis architecture interfaces and tools. In *Eur. Conf. Model Driven Softw. Eng.*, pages 1–11, Dec. 2003.

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

[17] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Int'l. Conf. Aut. Softw. Eng.*, pages 297–306, 2008.

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

[19] R. Majumdar and K. Sen. Hybrid concolic testing. In *Int'l. Conf. Softw. Eng.*, pages 416–426, 2007.

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

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

[22] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *J. Softw. Test., Verif., Rel.*, 9:263–282, Sept. 1999.

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

[24] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Prog. Lang. Des. Impl.*, June 2011.

[25] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Int'l. Symp. Softw. Test. Anal.*, pages 169–184, Aug. 1994.

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

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

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

[29] K. Taneja, T. Xie, N. Tillmann, J. Halleux, and W. Schulte. eXpress: Guided path exploration for regression test generation. In *Int'l. Symp. Softw. Test. Anal.*, July 2011.

[30] P. Tonella. Evolutionary testing of classes. In *Int'l. Symp. Softw. Test. Anal.*, pages 119–128, 2004.

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

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

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

[34] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Symp. on Found. Softw. Eng.*, Nov. 2010.

[35] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Asia-Pac. Softw. Eng. Conf.*, pages 406–413, Dec. 2009.