SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1562

Directed test suite augmentation: an empirical investigation

Zhihong Xu[1], Yunho Kim[2], Moonzoo Kim[2], Myra B. Cohen[1] and Gregg Rothermel[1,*,†]

[1]*Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE, USA*
[2]*Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, South Korea*

SUMMARY

Test suite augmentation techniques are used in regression testing to identify code elements in a modified program that are not adequately tested and to generate test cases to cover those elements. A defining feature of test suite augmentation techniques is the potential for reusing existing regression test suites. Our preliminary work suggests that several factors influence the efficiency and effectiveness of augmentation techniques that perform such reuse. These include the order in which target code 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 two empirical studies examining these factors, considering two test case generation algorithms (concolic and genetic). The results of our studies show that the primary factor affecting augmentation using these approaches 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 and in some cases a substantial effect on effectiveness. The order in which target code elements are considered turns out to have relatively few effects when using concolic test case generation but in some cases influences the efficiency of genetic test case generation. The results of our first study, on four relatively small programs using a large number of test suites, are supported by our second study of a much larger program available in multiple versions. Together, the studies reveal a potential opportunity for creating a more cost-effective hybrid augmentation approach leveraging both concolic and genetic test case generation techniques, while appropriately utilizing our understanding of the factors that affect them. Copyright © 2014 John Wiley & Sons, Ltd.

Received 10 May 2013; Revised 6 August 2014; Accepted 12 October 2014

KEY WORDS: regression testing; test case augmentation; concolic testing; genetic algorithms

1. INTRODUCTION

As software evolves, engineers regression test it to validate new features and detect whether new faults have been introduced into previously tested code. To help with this process, engineers often begin by reusing some or all of a system's previously developed test cases [1–3].

Reusing test cases is important, but having done so, engineers should also consider code or system behaviour, which, in the new version of the system, is not adequately tested. *Test suite augmentation* [4–7] helps with this, by identifying where new test cases are needed and creating them.

Despite the importance of improving test suites, most research on regression testing (Section 2) has focused simply on re-executing existing test cases. There has been research on approaches for identifying code that needs retesting [4, 6, 8–10], but these approaches do not then generate test cases, leaving that task to engineers. There has been research on automatically generating test cases given pre-supplied coverage goals (e.g. [5, 11–13]), but this research has not attempted to integrate the test case generation task with reuse of existing test cases in a regression testing context.

*Correspondence to: Gregg Rothermel, 256 Avery Hall, Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE 68588, USA.
†E-mail: grother@cse.unl.edu

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 (e.g. [13–15]), and such techniques can naturally leverage existing test cases.

To explore this belief, in a prior work [7], we developed a *directed test suite augmentation technique*. The technique begins by using a regression test selection algorithm [3] to identify code that should be retested (*target code*). The technique then uses existing test cases to seed a concolic test case generation algorithm [12] to create test cases that execute the target code. A case study showed that the approach was more efficient and better at covering target code than an analogous approach that did not utilize existing test cases. In subsequent work [16], we examined a similar approach to augmentation using a genetic algorithm for test case generation, also with promising results.

While these initial results were encouraging, our attempts to create augmentation techniques showed that several factors can potentially influence the efficiency and effectiveness of those techniques. The following factors are of particular interest, because they can be considered in the design of many types of augmentation techniques:

1. the order in which target code elements are considered (e.g. a random order or an order derived from a depth-first traversal of the flow graph of the program under test) while generating test cases,
2. the manner in which existing and newly generated test cases are used (e.g. the sample size used for reuse or the source of the sample) and
3. the algorithm (e.g. concolic or genetic) 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.

This article investigates this possibility. We designed and conducted two empirical studies investigating the foregoing factors in the context of test suite augmentation. Both of our studies consider concolic and genetic test case generation algorithms, two different orders of target code elements and two different manners of using existing test cases.

In our first empirical study, we consider relevant combinations of the foregoing factors on four programs for which a large number of test suites of varying ranges of initial coverage are available.[‡] We measure the effectiveness of the approaches in terms of code coverage and their efficiency in terms of the time required to perform augmentation. The results of this study 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 efficiency and effectiveness. The manner in which existing and newly generated test cases are utilized also has a substantial effect on efficiency and in some cases a substantial effect on effectiveness. The order in which target code elements are considered has relatively few effects when using concolic test case generation but does influence algorithm efficiency when using genetic test case generation. *Post hoc* analysis also shows that coverage characteristics (i.e. the degree of coverage exhibited by an initial test suite) can affect the impact of test reuse, but in our case, they did not affect the other factors studied.

In our second empirical study, we replicate the first on several versions of a substantially larger program, for which the existing test suite had different coverage characteristics than those used on the first set of programs. The results of our second study show that most of the effects observed in the first study extend to the larger program. However, the manner in which existing and newly generated test cases are reused has a much larger overall effect on the concolic algorithm, on this larger program, than it has on the genetic algorithm.

[‡]Results of an earlier version of this first study have been presented by Xu *et al*. [17]; this journal article is a revised and extended version of that paper, in which the study was re-executed under new experimental conditions, and a second, new study was conducted. Section 2 provides details.

DOI: 10.1002/stvr

Finally, an analysis of the coverage achieved by different approaches in both studies, and a qualitative analysis of coverage limitations exhibited by the two algorithms, reveals that the genetic and concolic approaches are often complementary in terms of the code they cover.

This work makes several contributions:

- We provide a new formalized algorithm for performing augmentation using various test case generation algorithms and settings of potentially influential factors.
- We report results on empirical studies comparing genetic and concolic test case generation in the context of augmentation and test reuse.
- Our results provide additional evidence that directed test suite augmentation techniques can be effective.
- Our results reveal factors that researchers and experimentalists may wish to consider when attempting to create and study directed test suite augmentation techniques.
- Our results reveal a potential opportunity for improving augmentation by employing hybrid approaches that leverage the different strengths of concolic and genetic algorithms, while appropriately utilizing our understanding of the factors that affect them.

The remainder of this article is organized as follows. Section 2 presents background and related work. Section 3 describes the test suite augmentation techniques that we consider and the factors relevant to them. Sections 4 and 5 present our two empirical studies, respectively. Section 6 provides further analysis of the differences observed between concolic and genetic algorithms in the context of different factors, culminating in a discussion of a potential hybrid algorithm. Finally, Section 7 presents our conclusions and discusses future work.

2. BACKGROUND AND RELATED WORK

In this section, we provide background and describe related work on test suite augmentation and automated test case generation.

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 techniques such as regression test selection (e.g. [3, 18–22]) and test case prioritization (e.g. [23–25]).

Test suite augmentation, in contrast, does not focus specifically on the reuse of T. Rather, it is concerned with the following tasks: (1) *identifying testing targets in* P' (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 cover or exercise these testing targets*.

Various algorithms have been proposed for identifying the code that should be tested in software systems following changes. Some of these approaches [26] operate on levels above the code such as on models or specifications, but most operate at the level of the code, and in this article, we focus on these. In practice, engineers often simply run existing test cases on the new system version, assess what code is not covered in that version by these test cases and then create test cases that cover this code. More sophisticated code-level techniques [8–10] use various analyses, such as slicing on program dependence graphs, to identify code elements that are potentially affected by changes and to select existing test cases that are related to these code elements. However, these approaches do not provide methods for generating new test cases that may be needed to cover the identified code.

Several recent papers [4–6, 27–31] specifically address test suite augmentation. Three of these papers [4, 6, 29] present an approach that combines dependence analysis and symbolic execution to identify chains of data and control dependencies, which, if tested, are likely to exercise the effects of changes. A potential advantage of this approach is a fine-grained identification of code elements requiring retesting; however, the papers do not present or consider any specific algorithms

DOI: 10.1002/stvr

for generating test cases. Person *et al*. [5] present an approach to program differencing using symbolic execution that can be used to identify testing targets more precisely than the approaches of Apiwattanapong *et al*. [4] and Santelices *et al*. [6, 29] and yields constraints that can be input to a solver to generate test cases for those requirements. In another work, Person *et al*. [27] use program analysis techniques to identify the parts of new programs that are affected by changes and apply symbolic execution only on these parts. Jin *et al*. [31] identify the parts of new programs that are affected by changes, generate test cases for the changed parts of new programs and report behavioural differences between the original and new programs. Qi *et al*. [30] present an approach for using dynamic symbolic execution to reveal execution paths that involve code changes and need to be retested, with the goal of making the effects of program changes observable. Finally, Taneja *et al*. [28] present an approach for using dynamic symbolic execution to prune irrelevant execution paths for regression testing.

In other related work [32], Yoo and Harman present a study of *test data augmentation*. They experiment with the quality of test cases generated from existing test suites using a heuristic search algorithm. While their work presents a technique that is similar to techniques that we consider in this article (because it uses a search algorithm seeded with existing test cases), their goal is to duplicate coverage in a single release in order to improve fault detection, not to obtain coverage of the target code elements in a subsequent release.

The test suite augmentation approach that we presented in earlier work [7] integrates a regression test selection technique [3] with an adaptation of the concolic test case generation approach [12]. This approach leverages test resources and data from prior testing sessions to perform both identification of coverage requirements and generation of test cases to cover these. The augmentation approach introduced by Xu *et al*. [16] operates similarly but uses a genetic algorithm to generate test cases. Case studies of the approaches showed that both can be effective and efficient. Both of these studies, however, were small, and neither study compared multiple augmentation approaches. Further, while the paper by Xu *et al*. [16] describes potentially influencing factors, it considers only one factor in detail.

This article is a revised and extended version of work presented in an earlier conference publication [17]. This article expands on that work in several ways:

- The conference paper presented a single experiment studying several augmentation techniques on four relatively small object programs. This article presents a new version of that experiment, which has been enhanced in two ways. First, we have used a new set of initial test suites that have a wider range of initial coverages, and this allows us to examine differences in algorithm performance that occur because of initial test suite characteristics. Second, we have compared augmentation techniques working with existing test suites with concolic test generation techniques working from scratch, and this allows us to consider the benefits of working with existing test cases.
- We have added a second empirical study on a larger object program available in several versions. This study provides evidence that the trends observed in the first study can generalize, while also showing the degree to which results may change owing to effects of scale.
- We have added additional analyses of data, including analyses of differences in technique performance on specific coverage targets overall and on several concrete cases. These analyses allow us to suggest ways in which a hybrid augmentation approach, combining the approaches studied in this paper, might achieve even greater cost-effectiveness.

2.2. Test case generation

While in practice, test cases are usually 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. [33–35]), from formal models (e.g. [36–38]), from semi-formal models (e.g. [39, 40]) and by random selection of inputs (e.g. [41, 42]).

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. [43–45]) use symbolic execution

DOI: 10.1002/stvr

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 [46] incorporate dynamic execution information into the search for inputs, using function minimization to solve subgoals that contribute towards an intended coverage goal. Goal-oriented techniques [14] also use function minimization to solve subgoals leading towards 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 (e.g. of data dependencies) to possibly influence progress towards that goal.

Several test case generation techniques use evolutionary or search-based approaches (e.g. [11, 13, 47–49]) such as genetic algorithms, tabu search and simulated annealing to generate test cases. Other works [12, 15, 50–52] combine 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 target source code. The approach has been extended to generate test data for database applications [53] and for Web applications using PHP [54, 55].

Implementations of several of the techniques discussed earlier are available. Java Path Finder [56] is a representative symbolic execution tool; it began as a software model checker but now is provided with various different execution models and extensions including some for generating test cases using symbolic execution. There are several tools (EXE [50], DART [15], CUTE [12], CREST [57], KLEE [58] and Microsoft Pex [59]) that apply concolic testing to unit testing of C programs. There are also tools that apply search-based techniques. For example, AUSTIN [60] is a structural test data generation tool (for unit tests) for the C language that uses search-based techniques. AUSTIN is based on the Common Intermediate Language framework and currently supports a random search, as well as a simple hill climber that is augmented with a set of constraint solving rules for pointer-type inputs. A second tool is called EvoSuite [61] and uses a hybrid approach for generating test cases for Java programs. This approach primarily uses a search-based approach but also integrates state-of-the-art techniques such as concolic testing and testability transformation.

Recently, researchers have attempted to combine test case generation techniques to improve testing effectiveness and efficiency. Inkumsah *et al*. [62] combine a genetic algorithm and concolic testing to generate test cases for object-oriented programs. Borges *et al*. [63] combine a meta-heuristic search technique and symbolic execution to solve complex mathematical constraints containing floating-point arithmetic. Symbolic search-based testing [64] uses symbolic execution to construct fitness functions that improve the efficiency of search-based testing for branch adequate test data generation. Malburg *et al*. [65] include constraint solving in the mutation stage of a genetic algorithm to generate mutated offspring that efficiently explores different execution paths in order to improve branch coverage. Galeotti *et al*. [66] consider the use of non-random starting points (provided by a genetic algorithm) in concolic testing. None of this work, however, has explicitly addressed the test case augmentation problem or studied the test case generation techniques in an augmentation context.

3. AUGMENTATION TECHNIQUES

We now describe the augmentation techniques that we consider in our empirical studies. We begin by presenting details relevant to the augmentation task as a whole, and then we present the specific algorithms that we utilize, along with details pertinent to their implementations.

3.1. Augmentation basics

3.1.1. Coverage criteria. In this work, we are interested in code-based augmentation techniques, and these typically involve specific code coverage criteria. In our studies, we focus on code coverage at the level of *branches*, that is, outcomes of predicate statements. We do this because branch coverage is stronger than statement coverage, but more tractable than criteria such as path-based coverage criteria, and hence more likely to be practical for and scale to larger systems.

DOI: 10.1002/stvr

3.1.2. Identifying target code elements. As noted in Section 1, test suite augmentation consists of two tasks: identifying the code that needs to be tested (*target code elements*) and creating test cases that exercise those code elements. As Section 2.1 shows, there are numerous approaches for identifying target code elements. In this work, we do not wish to compare these approaches, and we wish to be able to focus on factors affecting the augmentation task while limiting the possible confounding effects that may come from a particular choice of target code identification approaches. Thus, we choose a simple and practical approach for performing target code identification. Given program P and its test suite T and modified version P' of P, to identify target code 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 considered to be a target code 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 target code elements. Our augmentation techniques operate on lists of target code elements, and we believe that the order in which these elements are considered can affect the techniques because test cases covering one element may incidentally cover another.[§] In this work, we investigate the use of a depth-first order (DFO) of target code elements, but other orders are applicable.

The DFO of nodes in a control flow graph is the reverse of a postorder traversal of the graph [69, p. 660]. In data-flow analysis, considering nodes in 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 the analysis. The same approach can be applied to place branches in DFO. We conjecture that by considering target code elements in this order, we may achieve two things. First, we may be able to speed up the process of generating test cases, because test cases generated for target code elements that occur earlier in a program's control flow may incidentally cover elements occurring later in control flow, eliminating the need to specifically consider those later elements. Second, when using test case generation techniques that rely on constraint solvers, we may be able to achieve better efficiency by considering target code elements for which path constraints are shorter prior to those for which constraints are longer.

To apply this approach interprocedurally, we calculate DFO in terms of branches in a program's interprocedural control flow graph (ICFG) [70, 71]. We first build the ICFG, then we perform a postorder traversal of that graph by recording the branches visited and then we reverse the recorded order. Finally, we filter out branches that were not designated as target code elements to obtain our ordered list of target code elements.

For example, Figure 1 shows a simple ICFG. The E and X nodes represent method entry and exit. The numbered nodes represent statements in the program. The nodes with two outgoing edges represent predicate statements, and the labeled edges represent branches out of those predicates and the entry edges of methods (the latter ensures that the code in methods containing no branches is also covered). A postorder traversal of the graph visits branches in the order b7, b8, b3, b4, b1, b5, b9, b6, b2 and b0. The resulting DFO of the branches in the ICFG is thus b0, b2, b6, b9, b5, b1, b4, b3, b8 and b7. Considering branches in this order, we consider b7 only after we have considered b0, b1, b4 and b3. If we begin the augmentation process with just one test case that covers b0, b1, b3 and b8, we first filter out these four covered branches from the ordered list and then consider the remaining branches in the order b2, b6, b9, b5, b4 and b7.

3.1.4. Test case reuse approach. Existing test cases provide a rich source of data on potential program inputs and code reachability, and in the regression testing context, existing test cases are naturally available. We may have many existing test cases in an existing test suite to work with, and we may also generate many new test cases during the augmentation process. Choosing an effective subset of the existing test cases when attempting to cover a target element is important, because while larger subsets may provide more power for improving coverage, they also add to the cost of

[§]Consideration of element order is similar to the branch prioritization problem in concolic test case generation and has been included in prior work (e.g. [30, 67, 68]).

DOI: 10.1002/stvr

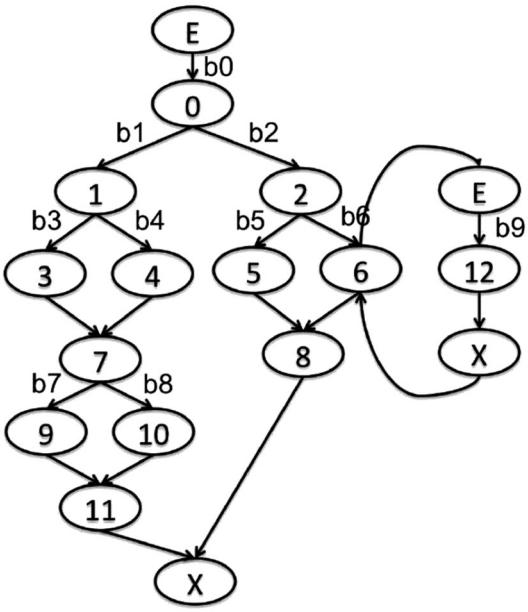

Figure 1. Interprocedural control flow graph.

the test case generation step. Two particular options for reusing test cases involve the following: (1) using only those test cases that existed initially; and (2) using test cases that existed initially together with any new test cases generated during augmentation. In our studies, we consider both of these approaches.

3.2. Augmentation algorithms

There are various test case generation algorithms that could be considered in the augmentation context. We chose to work with dynamic test case generation algorithms because these make use of existing test cases, and the ability to reuse such test cases is a central focus of this work. Among such algorithms, the most rigorously researched approaches to date involve concolic and search-based approaches, and among the latter, genetic approaches have been the most prevalent. Thus, it is these two approaches that we chose to investigate.

3.2.1. Main algorithm. Algorithm 1 controls the augmentation process, beginning with an initial set of existing test cases, *TC*; an ordered list of target code elements (target branches), B_{te}, which are not covered by the existing test cases *TC*; an iteration limit n_{iter}; and a Boolean flag *UseNew*.

The main loop (lines 5–22) continues until coverage no longer increases (which may occur owing to the iteration limit being reached in the test case generation routines). Within this loop, if b_t is not covered, a test case generation algorithm is called to generate test cases (line 9). If the algorithm 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).

To accommodate our other factor of concern—the manner in which existing and new test cases are used—we allow newly generated test cases to be added back into our set of available test cases. If Boolean flag *UseNew* is set to true, this causes the algorithm to do this (lines 17 and 18), 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 9 to generate new test cases.

3.2.2. 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 towards targets that can be blocks, branches or paths in a program [72–74]. To apply such an algorithm to a program, the test inputs must be represented in the form of a chromosome, and a fitness function

DOI: 10.1002/stvr

Input: set of existing test cases TC, ordered list of target code elements B_{te}, an iteration limit n_{iter} and a Boolean flag *UseNew*
Output: TC augmented with new test cases

```
foreach b_t ∈ B_te do
    Covered[b_t] = false
end
repeat
    NewCoverage = false
    for i = 1 to |B_te| do
        b_t = ith element of B_te
        if Covered[b_t] ≠ true then
            NewTests = AUGMENT(TC, B_te, b_t, n_iter)
            if NewTests ≠ Empty then
                NewCoverage = true
                B_cov = {All target code elements covered by NewTests}
                foreach b_cov ∈ B_cov do
                    Covered[b_cov] = true
                end
            end
            if UseNew then
                TC = NewTests ∪ TC
            end
        end
    end
until NewCoverage is false;
```

Algorithm 1: Main augmentation algorithm

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 we study in this work. The algorithm accepts four parameters: a set of test cases TC, a list of target code elements B_{te}, an uncovered target branch b_t, and an iteration limit n_{iter}. The algorithm returns a set of new test cases NTC, containing all generated test cases that cover previously uncovered branches in a modified program.

Input: set of test cases TC, list of target code elements B_{aff}, uncovered target branch $b_t \in B_{aff}$ and iteration limit n_{iter}
Output: set of new test cases NTC

```
NTC = ∅ // set of new test cases generated
TC_{b_t} = {test cases in TC that reach method m_{b_t}, the method containing b_t}
Population = TC_{b_t}
i = 0
repeat
    Fitness = CalculateFitness(Population)
    Population = Select(Population, Fitness)
    Population = Crossover(Population)
    Population = Mutate(Population)
    i = i + 1
    foreach tc ∈ Population do
        Execute (tc)
        UpdateCov(tc)
        if tc covers new branches in B_aff then
            NTC = NTC ∪ {tc}
        end
    end
until i ≥ n_iter or b_t is covered;
return NTC
```

Algorithm 2: Genetic augment algorithm

DOI: 10.1002/stvr

Instead of using randomly generated 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.

The algorithm repeats for a number of generations (set by the variable n_{iter}) or until b_t is covered. The first step (line 6) is to calculate the fitness of all test cases in the population. Because the fitness of a test case depends on its relationship to the branch targeted for coverage, calculating the fitness requires that the test case be executed. (For test cases provided initially, coverage information can be obtained while performing the prior execution of *TC*, which in our case occurred in conjunction with determining target code elements.) Next, a selection is performed (line 7), 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 9), 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, the algorithm is finished (line 18); otherwise, it iterates.

3.2.3. Concolic test suite augmentation. Concolic testing (concolic execution) [12, 15, 50] concretely executes a program while carrying along a symbolic state and simultaneously performing a 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 [12]. Conversely, symbolic execution helps generate concrete inputs for the next execution to increase coverage in the concrete execution scope.

In traditional concolic testing, test case reuse is not considered, and test case generation focuses 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 the algorithm then iterates in an attempt 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 3.2.1; this includes leveraging existing test cases and operating on an ordered list of target code 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 (actually, *branches* in P) between N_P.
- A *path condition pc* of a target program P is a conjunction $b_{i_1} \wedge b_{i_2} \wedge \ldots b_{i_n}$ where $b_{i_1}, \ldots, b_{i_n}$ are the conditions associated with taking given branches in E_P and are executed in order. Note that n can be larger than $|E_P|$, as a condition associated with a 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 the condition associated with a branch occurring at the jth position in pc and removing all subsequent conditions associated with 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, then $\overline{b}$ is the `else` branch).
- $LastPos(b, pc)$ returns a last position j of a condition associated with a branch b_{i_j} in a path condition pc where $b = b_{i_j}$ (i.e. $\forall j < k \leq n \cdot b_{i_k} \neq b$). $LastPos(b, pc)$ returns 0 if b does not exist in pc.
- $Solve(pc)$ returns a test case satisfying the path condition pc if pc is satisfiable; UNSAT otherwise.

DOI: 10.1002/stvr

We assume that conditional statements always have two branches for the sake of simplicity; `switch` statements can be transformed into equivalent nested `if` statements (Section 4.3.2).

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. The beginning of the main procedure resets the set of newly generated test cases *NTC* (line 1) and selects test cases that can reach $\overline{b_t}$ (the paired branch of b_t) from *TC* (line 2). If there are no such test cases, the algorithm terminates (lines 3 and 4). If there are such test cases, the algorithm obtains path conditions by executing the target program with the selected test cases (line 6). Suppose the last occurrence of $\overline{b_t}$ is located in the mth condition associated with a branch of pc. From each path condition pc obtained, the algorithm generates the minimum of n_{iter} and m new path conditions as follows. The algorithm generates n_{iter} new path conditions (lines 8–20) by negating $b_{i_m}, b_{i_{m-1}}, \ldots, b_{i_{m-n_{iter}+1}}$ and removing all following conditions associated with branches in pc, respectively (line 10). If a newly generated path condition pc' has a solution tc_{new} (a new test case) (lines 11 and 12) and tc_{new} covers uncovered branches in B_{te} (line 15), tc_{new} is added to *NTC* (line 16).

```
   Input: set of test cases TC, list of target code elements B_te, an uncovered target branch b_t ∈ B_te and an iteration
          limit n_iter
   Output: set of new test cases NTC
1  NTC = ∅  // new test cases
2  TC_b̄t = {all test cases in TC that reach b̄t}
3  if TC_b̄t = ∅ then
4  |  return ∅
5  end
6  PC_b̄t = {path conditions obtained from executing test cases in TC_b̄t}
7  foreach pc ∈ PC_b̄t do
8  |  for i = LastPos(b̄t, pc) to i − n_iter + 1 do
9  |  |  if i > 0 then
10 |  |  |  pc' = DelNeg(pc, i)
11 |  |  |  tc_new = Solve(pc')
12 |  |  |  if tc_new ≠ UNSAT then
13 |  |  |  |  Execute(tc_new)
14 |  |  |  |  UpdateCov(tc_new)
15 |  |  |  |  if tc_new covers uncovered branches in B_te then
16 |  |  |  |  |  NTC = NTC ∪ {tc_new}
17 |  |  |  |  end
18 |  |  |  end
19 |  |  end
20 |  end
21 end
22 return NTC
```

Algorithm 3: Concolic augment algorithm

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 can affect both the efficiency and the effectiveness of the approach.

4. EMPIRICAL STUDY 1

Our goal is to investigate augmentation techniques focusing on three factors (test case generation algorithm, order of target code elements and test reuse approach). We thus pose the following research questions.

RQ1: How does the order of consideration of target code elements affect augmentation techniques?
RQ2: How does the manner of use of existing and newly generated test cases affect augmentation techniques?
RQ3: How does the use of genetic and concolic test case generation techniques affect augmentation techniques?

DOI: 10.1002/stvr

Table I. Objects of analysis.

Program	Functions	Lines of code	Branches	Test cases
printtok1	21	402	174	3052
printtok2	20	483	186	3080
replace	21	516	206	3174
tcas	8	138	76	1608

Table II. Branch coverage and sizes of initial test suites.

	Branch coverage			Test suite size		
Program	Average	Minimum	Maximum	Average	Minimum	Maximum
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
tcas	57.9	30.0	69.0	10.8	5	16

4.1. Objects of analysis

To facilitate technique comparisons, our objects of analysis (programs and test suites) must be suitable for use by both implementations. To select appropriate objects, we examined C programs available in the Software–artifact Infrastructure Repository [75]. We selected four programs[¶] (Table I), each of which is available with a large 'universe' of test cases; these test cases were created with the aim of achieving both requirements and code coverage of the programs and with the aim of including numerous test cases (a minimum of 30) for each coverable requirement or code element [76].[‖]

The programs that we selected do not have actual sequential versions. 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 programs. This let us model a situation where prior test suites are inadequate and require augmentation.

To create such test suites, we did the following. First, for each 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. (We chose 1000 because it is a number beyond which—on all programs—further increases fail to lead to changes in observed minimum and maximum sizes.) 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 obtained by these test suites are given in Table II. (Branch coverage numbers listed in the table are for the extended programs (explained further in Section 4.3.3)).

[¶]For this study, we began by considering the seven Siemens programs, because their size is amenable to study on enormous numbers of test cases. Constraint solvers, however, have limitations, and available satisfiability modulo theory solvers do not handle floating-point arithmetic, which is present in the three Siemens programs not selected.

[‖]Concolic test case generation techniques set limits on the sizes of inputs they generate, and some inputs in the test pools provided with the programs did not conform to reasonable limits. We thus ran several trials with various size limits and selected limits that let us retain at least 60% of the inputs in the original test universes. We then removed, from the test universes, test cases that did not conform to these limits. Table I lists the sizes of the test universes after this reduction.

DOI: 10.1002/stvr

4.2. Variables and measures

4.2.1. Independent variables. Our experiment manipulated three independent variables:

IV1: Order in which target code elements are considered. As orders of target code elements, we use the DFO described in Section 3 and a baseline approach that orders target code 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 use 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 Section 3.[**]

4.2.2. Dependent variables and measures. 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. There are various approaches that could be used to measure technique effectiveness. One approach involves assessing the ability of techniques to detect faults present in systems. This approach has the advantage of directly measuring a quality of test suites that is typically most sought after by test engineers. From an empirical standpoint, however, measuring fault-detection effectiveness has drawbacks. The fault detection ability of a testing technique can vary depending on the number and type of faults present in systems, the locations of faults, and the types of test oracles used [77]. It is possible, then, that sets of faults present in our object programs might be (inadvertently) more prone to being detected by particular classes of test case generation techniques than by others. This is a serious threat to internal validity, because it could lead us to conclude, erroneously, that one technique is more effective than another.

A second approach that has been used to assess the effectiveness of test suite augmentation techniques involves tracking behavioural differences (such as differences in outputs) [29–31]. While this is appropriate for assessing techniques that have a goal of exposing behavioural differences between program versions, it is not appropriate for assessing techniques, such as ours, that have a goal of improving test coverage. This is because not all test cases that improve coverage necessarily expose behavioural differences, and not all test cases that expose behavioural differences necessarily improve coverage.

A third approach to assessing technique effectiveness involves the use of data on the code coverage achieved by test cases. While this approach does not rely on faults or measures of behavioural differences, it has several advantages in the context of this study. First, coverage does not possess problems related to locations and types of faults that can affect internal validity. Second, the test case augmentation techniques that we consider are directly intended to work with existing test suites to achieve higher levels of coverage in a modified program P, so measuring coverage is a direct measure of technique intent and treats all of the techniques that we study in similar manners. Finally, there is substantial evidence in the testing literature that code coverage can correlate with fault detection. This correlation has been reported directly in relation to the use of code coverage criteria [76, 78–81]. The correlation has also been observed indirectly in the fact that coverage-based test case prioritization techniques, which order test cases in manners that are intended to detect faults faster, do in fact succeed in doing so even when using orderings based solely on simple coverage metrics (e.g. [23, 82, 83]).

We thus selected code coverage as our effectiveness measure. To measure the effectiveness of our techniques, we track the number of branches in each object program that can be covered by each augmented test suite.

[**] While it is possible to use many different variants of these algorithms and to experiment with differing parameters, fixing our implementations as much as possible allows us to isolate the effect of other factors more effectively.

DOI: 10.1002/stvr

DV2: Efficiency. 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 it.

4.3. Experiment set-up

Several steps had to be followed to establish the experiment set-up needed to conduct our study.

4.3.1. Genetic algorithm implementation. We implemented the algorithm presented in Section 3; however, doing this involved several implementation decisions. In making these decisions, we chose to focus on creating a simple implementation; this lets us avoid overtuning the algorithm for the specific problems.

First, in our case, each test case is a chromosome where the genes are inputs to the programs, and we customized this for each program. For example, for `printtok1` and `printtok2`, all the characters in the input file form a chromosome, and each character is a gene, while for `tcas`, each integer in the input is a gene in the chromosome. For selection, we selected the best half of the population to generate the next generation; we kept the selected chromosomes in the new generation. We ranked the chromosomes and divided them into two parts. The first chromosome in the first half was mated with the first chromosome in the second half, the second chromosome in the first half was mated with the second chromosome in the second half and so forth.

Second, in crossover, we performed a one-point crossover by randomly selecting a position that is between 0 and the number of genes of the smaller chromosome; we then swapped everything between chromosomes starting at that position to the end of the chromosome. In performing this process, we did not attempt to preserve the well-formedness of the input. Existing test cases were almost all well formed, so newly generated test cases that were not well formed helped increase coverage.

Third, our search targets are branches in the program; therefore, for our fitness function, we used the approach level described by Wegener *et al*. [84]. We chose the traditional form of a fitness function that targets a single branch at a time, over more recent implementations (such as EvoSuite) that target all branches at once. We did this for two reasons. First, it matches the method used by the concolic algorithm. Second, we are testing at the system level (rather than the unit level), and the number of branches in whole programs is substantially larger than the numbers of branches in individual units. We found during initial experiments that calculating fitness based on coverage of all branches was much slower than targeting a single branch at a time. Finally, because we are interested in augmentation (rather than in generating test cases to cover all branches), our target pool is only a small subset of the full branch set. The approach level is based on control dependence distance; it is a discrete count measuring how far we are from the predicate controlling the target branch in the control dependence graph when we deviate course. The further away we are from this predicate when we take the wrong control dependence branch, the higher the value will be. Thus, we want to minimize this value as we search. For instance, if we reach the predicate leading to our target, the approach level is 0. If we take the wrong branch at the node just above this predicate, the approach level is 1.[††]

Finally, we attempted to minimize our tuning between programs to avoid bias in our experiments, but we found that the mutation rate was critical for convergence in the individual programs. Therefore, we used different mutation rates that were obtained heuristically through observation. For `printtok1` and `printtok2`, we used 0.06; for `replace`, we used 0.08; and for `tcas`, we used 0.05.

[††]In our initial implementation, for the sake of simplicity and because of instrumentation overhead, we did not combine approach level with branch distance. We collected data relative to early implementations and did not observe any improvements when branch distance was added. Because we nonetheless achieved good convergence on these programs, we did not consider branch distance as part of our approach. Still, research suggests that branch distance can be an important part of fitness functions [85], so we intend to consider it further in the future.

DOI: 10.1002/stvr

4.3.2. Concolic algorithm implementation. To implement the concolic test case generation algorithm presented in Section 3, we created a tool based on CREST [57, 67]. 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()}`). In addition, CREST transforms `switch` statements into nested conditional statements (e.g. `switch(c){ case 1: f();break; case 2: g();break;}` is transformed into `if (c==1) f(); else if (c==2) g();`).

4.3.3. Extended programs. To facilitate fair comparisons between concolic and genetic algorithms, 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. All results, including all measures of coverage, are reported relative to these extended versions.

4.3.4. 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 efficiency 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 between 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 3.2.2 and 3.2.3.)

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 target code element orders (random or depth first) and M is one of the two test case reuse approaches (existing test cases or new + existing test cases). An individual *augmentation technique application* consists of an augmentation technique applied at an iteration limit L, and in our case, L has five levels.

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

Our experiments were run on Linux boxes with Intel Core2Duo E8400s at 3.6 GHz and with 16 GB RAM, running Fedora 9 as operating systems. Our processes were the only user processes active on the machines.

4.5. Threats to validity

The primary class of threats 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 the study of other programs and other types of code changes may exhibit different cost-benefit trade-offs. While the universes of test cases provided with these programs were not randomly generated (test cases were deliberately designed to target requirements and code components), we did randomly select test cases from these universes to create test suites; test suites generated by other mechanisms may also exhibit different trade-offs. Furthermore, our programs are chosen to allow the application of both genetic and concolic testing and, thus, do not reveal cases in which program characteristics might hinder one but not the other of these approaches.

A second class of threats to external validity pertains to our algorithms. We have utilized only one variant of a genetic test case generation algorithm with specific choices for selection, crossover, and fitness calculation operations, and we have used only one variant of a concolic testing algorithm.

DOI: 10.1002/stvr

Genetic algorithms require tuning in terms of fitness function, selection method, and mutation mechanism. We performed minimal tuning in order to avoid overfitting, but alternative tunings might have allowed the genetic algorithms to perform differently. Similarly, we have used just one concolic algorithm and implementation, and alternative algorithms or implementations might allow the approach to perform differently. Finally, as we have also mentioned, efficiency differences between the implementations cannot be compared in any rigorously quantitative sense.

In addition to the foregoing threats, we have applied both algorithms to extended versions of the programs, where the genetic approach does not require this and might function differently on the original source code. Further, we have considered only two approaches for handling target branches; other approaches or approaches that handle sets of target branches rather than single branches (e.g. [86]) may exhibit different results.

Threats to external validity such as the foregoing can be addressed only by additional studies.

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

Where *construct validity* is concerned, there are other metrics that could be pertinent to the effects studied. Our effectiveness measurements focus on code coverage, and studies of fault-detection effectiveness may yield additional insights. Further, 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 in our discussion of internal validity, given the inherent difficulty of obtaining efficient technique prototypes.

Where *conclusion validity* is concerned, our choices of iteration limits for the two test case generation algorithms may have limited our ability to compare the genetic and concolic algorithms fairly in regard to RQ3; it is possible that the addition of additional levels could alter the results of the comparison.

4.6. Results and analysis

As an initial overview of the data, Tables III–VI present data on final coverage and increased coverage (final coverage − initial coverage), and Tables VII–X present data on cost. Final coverage, increased coverage and cost values are obtained per program, across all test suites, for each iteration limit and for each combination of order of target code elements and test reuse approach. Coverage is shown as the number of branches covered by augmented test suites, and cost is shown in seconds. Each table presents results for concolic and genetic techniques for one combination of the branch order and test case reuse treatments.

Table III. Coverage using depth-first order and existing test cases.

	Total branches	Initial coverage	Final coverage					Increased coverage				
Genetic			5	10	15	20	25	5	10	15	20	25
`printtok1`	174	133.3	154.9	155.9	156.0	156.3	156.7	21.6	22.6	22.7	23.0	23.4
`printtok2`	186	158.8	175.9	176.2	176.3	176.4	176.4	17.1	17.4	17.5	17.6	17.6
`replace`	206	165.9	184.6	185.4	186.3	186.7	186.9	18.7	19.5	20.4	20.8	21.0
`tcas`	76	57.9	69.7	70.5	70.7	70.8	70.8	11.8	12.6	12.8	12.9	12.9
Concolic			1	3	5	7	9	1	3	5	7	9
`printtok1`	174	133.3	144.0	150.4	151.3	152.2	152.5	10.7	17.1	18.0	18.9	19.2
`printtok2`	186	158.8	165.5	170.8	171.8	172.5	173.2	6.7	12.0	13.0	13.7	14.4
`replace`	206	165.9	176.5	185.6	188.3	189.2	189.6	10.6	19.7	22.4	23.3	23.7
`tcas`	76	57.9	65.1	66.8	68.9	69.5	69.5	7.2	8.9	11.0	11.6	11.6

Unit: number of branches.

DOI: 10.1002/stvr

Table IV. Coverage using depth-first order and existing plus new test cases.

	Total branches	Initial coverage	Final coverage					Increased coverage				
Genetic			5	10	15	20	25	5	10	15	20	25
`printtok1`	174	133.3	155.7	156.8	157.0	157.4	157.8	22.4	23.5	23.7	24.1	24.5
`printtok2`	186	158.8	176.4	176.6	176.5	176.6	176.6	17.6	17.8	17.7	17.8	17.8
`replace`	206	165.9	185.2	186.5	186.7	187.3	187.2	19.3	20.6	20.8	21.4	21.3
`tcas`	76	57.9	70.7	70.9	70.9	71.0	71.0	12.8	13.0	13.0	13.1	13.1
Concolic			1	3	5	7	9	1	3	5	7	9
`printtok1`	174	133.3	144.2	150.7	151.7	152.5	152.9	10.9	17.4	18.4	19.2	19.6
`printtok2`	186	158.8	165.7	171.3	172.2	172.9	173.7	6.9	12.5	13.4	14.1	14.9
`replace`	206	165.9	176.8	187.5	189.7	190.5	190.8	10.9	21.6	23.8	24.6	24.9
`tcas`	76	57.9	65.6	67.7	70.2	70.9	70.9	7.7	9.8	12.3	13.0	13.0

Unit: number of branches.

Table V. Coverage using random order and existing test cases.

	Total branches	Initial coverage	Final coverage					Increased coverage				
Genetic			5	10	15	20	25	5	10	15	20	25
`printtok1`	174	133.3	154.9	155.5	155.6	156.2	156.4	21.6	22.2	22.3	22.9	23.1
`printtok2`	186	158.8	175.7	176.3	176.5	176.5	176.4	16.9	17.5	17.7	17.7	17.6
`replace`	206	165.9	184.4	185.9	186.3	186.9	187.0	18.5	20.0	20.4	21.0	21.1
`tcas`	76	57.9	69.8	70.6	70.7	70.8	70.8	11.9	12.7	12.8	12.9	12.9
Concolic			1	3	5	7	9	1	3	5	7	9
`printtok1`	174	133.3	144.0	150.4	151.3	152.2	152.5	10.7	17.1	18.0	18.9	19.2
`printtok2`	186	158.8	165.5	170.8	171.8	172.5	173.2	6.7	12.0	13.0	13.7	14.4
`replace`	206	165.9	176.5	185.6	188.3	189.2	189.6	10.6	19.7	22.4	23.3	23.7
`tcas`	76	57.9	65.1	66.8	68.9	69.5	69.5	7.2	8.9	11.0	11.6	11.6

Unit: number of branches.

Table VI. Coverage using random order and existing plus new test cases.

	Total branches	Initial coverage	Final coverage					Increased coverage				
Genetic			5	10	15	20	25	5	10	15	20	25
`printtok1`	174	133.3	155.6	156.2	156.7	157.7	157.3	22.3	22.9	23.4	24.4	24.0
`printtok2`	186	158.8	176.5	176.6	176.5	176.6	176.6	17.7	17.8	17.7	17.8	17.8
`replace`	206	165.9	185.4	186.3	187.0	187.6	187.4	19.5	20.4	21.1	21.7	21.5
`tcas`	76	57.9	70.6	70.9	70.9	70.9	71.0	12.7	13.0	13.0	13.0	13.1
Concolic			1	3	5	7	9	1	3	5	7	9
`printtok1`	174	133.3	144.2	150.7	151.7	152.4	152.8	10.9	17.4	18.4	19.1	19.5
`printtok2`	186	158.8	165.7	171.3	172.2	172.9	173.7	6.9	12.5	13.4	14.1	14.9
`replace`	206	165.9	176.8	187.6	189.5	190.6	190.8	10.9	21.7	23.6	24.7	24.9
`tcas`	76	57.9	65.6	67.2	70.2	70.9	70.9	7.7	9.3	12.3	13.0	13.0

Unit: number of branches.

We now discuss and analyse these data with respect to our three research questions, in turn.

4.6.1. RQ1: order of target code elements. Our first research question pertains to the effects of using different orders of target code elements; in this case, DFO versus random. Table XI presents a view of our data that helps us address this question. The table presents results per program, with

DOI: 10.1002/stvr

Table VII. Cost using depth-first order and existing test cases.

	Cost (s)				
Genetic	5	10	15	20	25
printtok1	38.7	78.9	117.2	158.4	194.6
printtok2	26.2	54.9	83.9	113.1	151.7
replace	65.7	128.4	185.2	247.5	322.7
tcas	3.1	5.6	8.3	11.1	13.7
Concolic	1	3	5	7	9
printtok1	1.6	4.4	7.1	9.9	12.6
printtok2	0.3	0.5	0.8	1.1	1.4
replace	0.9	2.9	5.0	6.9	9.0
tcas	0.1	0.2	0.3	0.4	0.4

Table VIII. Cost using depth-first order and existing plus new test cases.

	Cost (s)				
Genetic	5	10	15	20	25
printtok1	81.2	151.3	239.5	314.8	385.7
printtok2	54.5	106.3	147.4	229.0	272.6
replace	92.3	183.3	283.4	365.8	449.6
tcas	5.1	9.6	14.3	18.7	24.3
Concolic	1	3	5	7	9
printtok1	1.9	5.7	9.4	13.1	16.6
printtok2	0.3	0.6	0.9	1.3	1.6
replace	1.1	3.9	6.7	9.3	11.9
tcas	0.1	0.2	0.3	0.4	0.5

Table IX. Cost using random order and existing test cases.

	Cost (s)				
Genetic	5	10	15	20	25
printtok1	81.2	151.3	239.5	314.8	385.7
printtok2	54.5	106.3	147.4	229.0	272.6
replace	92.3	183.3	283.4	365.8	449.6
tcas	5.1	9.6	14.3	18.7	24.3
Concolic	1	3	5	7	9
printtok1	1.9	5.7	9.4	13.1	16.6
printtok2	0.3	0.6	0.9	1.3	1.6
replace	1.1	3.9	6.7	9.3	11.9
tcas	0.1	0.2	0.3	0.4	0.5

coverage results in the upper half and cost results in the bottom half. Column headers use mnemonics to indicate techniques: GDE corresponds to (genetic, DFO, existing), GDN to (genetic, DFO, new + existing), GRE to (genetic, random, existing), GRN to (genetic, random, new + existing), CDE to (concolic, DFO, existing), CDN to (concolic, DFO, new + existing), CRE to (concolic, random, existing) and CRN to (concolic, random, new + existing). Individual columns correspond to techniques compared; thus, column 2, with header 'GDE versus GRE', compares (genetic, DFO, existing) with (genetic, random, existing).

Each entry in the table summarizes the differences observed between the two techniques for each of the five iteration limits. 'D' indicates that the technique using DFO exhibited the better (greater)

DOI: 10.1002/stvr

Table X. Cost using random order and existing plus new test cases.

	Cost (s)				
Genetic	5	10	15	20	25
printtok1	89.3	171.1	248.6	379.7	428.5
printtok2	64.9	114.8	165.7	201.2	294.7
replace	93.5	188.6	285.2	375.8	471.0
tcas	5.2	9.7	15.1	20.6	25.3
Concolic	1	3	5	7	9
printtok1	1.9	5.7	9.4	12.9	16.5
printtok2	0.3	0.6	0.9	1.3	1.6
replace	1.1	4.0	6.8	9.5	12.2
tcas	0.1	0.2	0.3	0.4	0.5

Table XI. Impact of order in which target code elements are considered on coverage and cost.

	GDE versus GRE	GDN versus GRN	CDE versus CRE	CDN versus CRN
		Coverage		
printtok1	R D D D D	D D D R D	= = = = =	= = D D D
printtok2	D R ***R*** R =	R = R R R	= = = = =	= R R R R
replace	D R = D R	R D R R R	= = = = =	R R D R R
tcas	R R D D =	D D R D R	= = = = =	D ***D*** = = R
		Cost		
printtok1	***D D D D D***	***D D*** D ***D D***	R R R D ***D***	R R R ***R*** R
printtok2	***D D D D D***	***D D D*** R ***D***	D R R R R	R D D R R
replace	***D D D D*** R	***D D D D D***	R ***D D D D***	R D ***D D D***
tcas	R R R ***D D***	***D D D D*** D	= R R D R	D D R D D

mean coverage value or better (lesser) cost value, 'R' indicates that the technique using random order exhibited the better (greater) mean coverage or better (lesser) cost value and '=' indicates that techniques exhibited equal mean coverage or cost (through the second decimal place). For example, for `printtok1`, comparing GDE and GRE for coverage, the table contains 'R D D D D', indicating that at the lowest iteration limit, random order produced greater coverage, and at the other four limits, DFO produced greater coverage. A similar entry for `printtok1` for cost, containing 'D D D D D', indicates that at all five iteration limits, DFO exhibited the lowest cost.

For each pair of techniques compared, for each iteration limit L, we applied a *Wilcoxon–Mann–Whitney* [87] test to the coverage (cost) data obtained across all test suites augmented using $\alpha = 0.05$ as the significance threshold, to validate the null hypothesis: there is no significant difference between two orders (DFO and random) in terms of effectiveness (efficiency) when corresponding techniques are compared at iteration limit L. (We did not use a *t-test* because our data are not normally distributed.) In the table, bold italicized fonts indicate statistically significant differences. For example, for `printtok2`, comparing GDE and GRE for coverage, the only statistically significant difference between techniques occurred at iteration limit 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 resided in either test case order, and results were relatively similar in the instances in which existing, or new and existing, test cases were used. Across all iteration limits and programs, there was only one case in which the two orders resulted in a statistically significant difference (`printtok2` at iteration limit 15). Even when considering the non-statistically significant differences between orders, there was no clear winner.

Where cost results for the genetic algorithm are concerned, we see different trends. First, in the GDE-versus-GRE case, there were 16 instances in which order caused statistically significant

DOI: 10.1002/stvr

differences: these included all instances for `printtok1` and `printtok2` and most instances for `replace`. In the GDN-versus-GRN case, there were also 17 instances, again including all instances for `replace` and most instances for `printtok1` and `printtok2`. In all of these instances, DFO was less costly than random.

Turning to the concolic approach, where coverage is concerned, when new and existing test cases are considered, there is only one statistically significant difference between techniques, for `tcas` in the CDN-versus-CRN case at iteration limit 3. Again, even non-statistically significant differences revealed no clear winner. Moreover, when only existing test cases were used, techniques exhibited no differences in coverage at all. Therefore, there were no apparent patterns involving iteration limits or programs to indicate that order influenced coverage.

Finally, considering cost results for the concolic approach, unlike the case for the genetic approach, we found only a few statistically significant differences in costs, with five in the CDE-versus-CRE case and four in the CDN-versus-CRN case. Seven of these instances were on `replace`, where DFO was less costly than random, and these were at higher iteration limits, so this may indicate some trend that will emerge as programs become more complex. However, for the other three programs, there was no clear advantage adhering to either random or DFO orders.

4.6.2. 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 XII presents data relevant to this question. The table format is similar to that of Table XI, but in keeping with the goal of comparing across test case reuse approaches, the differences in terms compared all involve reuse approaches (existing vs new + existing). 'E' indicates cases in which the use of only existing test cases was superior, and 'N' indicates cases in which the use of new plus existing test cases was superior. Bold italicized fonts indicate statistically significant differences. For each pair of techniques compared, for each iteration limit L, we again applied a *Wilcoxon–Mann–Whitney* [87] test to the coverage (and cost) data obtained across all test suites augmented using $\alpha = 0.05$ as the significance threshold, to validate the null hypothesis: there is no significant difference between the two methods of reusing test cases in terms of effectiveness (efficiency) when corresponding techniques are compared at iteration limit L.

We begin by considering the results for the genetic algorithm. Where coverage is concerned, in all instances, the use of new and existing test cases was superior to reusing only existing test cases, and in most instances, the difference was statistically significant. This includes 19 of 20 instances when DFO was used and 16 of 20 instances in which random order was used.

Where cost results for the genetic algorithm are concerned, we observe even stronger effects: in all instances, using existing test cases only was less expensive, and the effect of doing so was statistically significant.

Turning to the concolic approach, where coverage is concerned, here, we see a strong evidence that test case reuse mattered for coverage, with the use of new and existing test cases always more effective, and in all instances statistically significantly so.

Table XII. Impact of test case reuse approaches on coverage and cost.

	GDE versus GDN	GRE versus GRN	CDE versus CDN	CRE versus CRN
		Coverage		
`printtok1`	***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***
`replace`	***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 N N N***	***N N N N N***	***N N N N N***
		Cost		
`printtok1`	***E E E E E***	***E E E E E***	***E E E E E***	***E E E E E***
`printtok2`	***E E E E E***	***E E E E E***	***E E E E E***	***E E E E E***
`replace`	***E E E E E***	***E E E E E***	***E E E E E***	***E E E E E***
`tcas`	***E E E E E***	***E E E E E***	***E E E E E***	***E E E E E***

DOI: 10.1002/stvr

Finally, considering cost results for the concolic approach, we again note statistically significant differences in all instances, again with lower costs adhering to the use of only existing test cases.

4.6.3. 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 them for 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 the concolic approach and a given iteration limit for the genetic approach; 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 located the iteration limit L_g at which the genetic algorithm operated most effectively on P and the iteration limit L_c at which the concolic algorithm operated most effectively on P, and we compared the algorithms at these respective optimal iteration limits. To perform these comparisons, we again applied a *Wilcoxon–Mann–Whitney* [87] test to the coverage data at the chosen iteration limits using $\alpha = 0.05$ as the significance threshold, to validate the null hypothesis: there is no significant difference between the effectiveness of the two test case generation techniques.

Table XIII presents data relevant to RQ3 with respect to algorithm effectiveness following the analysis procedure just described. The table provides data for each program and for each of the four combinations of target code element order and test reuse strategies studied. An individual table entry indicates which technique achieved greater coverage ('G' indicating the genetic algorithm and 'C' indicating the concolic algorithm), and bold italicized fonts indicate instances in which the difference is statistically significant.

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

Turning to efficiency, this comparison is complicated by the inherent differences in our two implementations. In fact, it is quite difficult to fairly compare genetic and concolic techniques for efficiency because their implementations are derived from different sources and 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 VII–X show, costs for the genetic algorithm ranged from times in the tenths of seconds to times above 400 s, while costs for the concolic algorithm ranged from times in the tenths of seconds to times near 20 s. With our current implementations, this represents a very large difference in favour of the concolic approach.

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

Table XIII. Comparison of coverage achieved by test case generation techniques: genetic versus concolic.

Program	GDE versus CDE	GDN versus CDN	GRE versus CRE	GRN versus CRN
`printtok1`	***G***	***G***	***G***	***G***
`printtok2`	***G***	***G***	***G***	***G***
`replace`	***C***	***C***	***C***	***C***
`tcas`	***G***	***G***	***G***	***G***

DOI: 10.1002/stvr

4.7. Discussion and implications

We now discuss the results presented in the prior section and provide further analysis.

4.7.1. Target code element order. One might argue that in principle, the order of target code elements is not likely to significantly affect algorithm effectiveness in terms of coverage achieved, because the same elements will ultimately be considered under any order. This is what we observed in the results of our study.

Where efficiency is concerned, in contrast, we did observe differences: our results show that DFO provided savings in costs when using the genetic algorithm. This can be explained by observing that with the genetic algorithm, if we target higher-level branches first, we can incidentally cover additional branches. Also, test cases that cover branches higher in dependency chains could 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 were mixed. We suspect this is because test cases generated to cover a given branch b_t (lines 11–19 of Algorithm 3) may fail to cover other uncovered branches unless these uncovered branches share a common ancestor branch at a short distance from b_t (less than n_{iter}) in an execution tree. In such cases, the order of target code elements is not likely to affect cost.

All things considered, based on our results, we conjecture that DFO has the potential to be more efficient than random order when using genetic algorithms, because we observed this in almost all cases considered. There appears to be no clear benefit to using either order, however, where the concolic approach is concerned. Still, these results do not preclude finding some other orders that are more predictably cost-effective for that approach.

4.7.2. Test case reuse approach. Our results show that the use of new test cases in addition to existing test cases always significantly increased the cost of test case 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 affects the population size for 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 case generation technique effectiveness in all scenarios in which the concolic approach was used, and in most scenarios in which the genetic approach was used. The difference across techniques 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 have been used previously. The concolic approach, in contrast, utilizes each new test case independently and can potentially gain from each as such.

If these results generalize, we have identified an important cost-benefit trade-off. With both test case generation techniques, we found a potential pay-off for incurring the additional costs involved in reusing test cases, and this effect was greater for the concolic technique than for the genetic technique. In practice, whether any effectiveness gain is worth the additional cost must be assessed relative to the actual costs of generating test cases versus the actual benefits of obtaining better coverage on the particular systems being verified. Such assessments, however, are viable in the context of software evolution, where systems are expected to be retested many times, and long-term cost-benefit gains make assessments more worthwhile.

4.7.3. Test case generation techniques. As mentioned in our discussion of threats to validity, we are studying particular variants and implementations of test case generation algorithms. Keeping that discussion in mind, in our experiment, the concolic and genetic test case generation techniques that we studied did perform statistically significantly differently. The genetic algorithm exhibited greater effectiveness than the concolic algorithm on `printtok1`, `printtok2` and `tcas` under all combinations of other factors. It appears that the genetic algorithm was more costly (potentially by two orders of magnitude) than the concolic algorithm in doing this, although again, this comparison

DOI: 10.1002/stvr

must be made cautiously. These observations do prompt us, however, to further explore the reasons for differences. We postpone discussion of that exploration to Section 6, however, when we can present it together with further input from the results of our second study.

4.7.4. Iteration limits. Our focus in this work is on test case generation algorithms, test case reuse approaches and target code element orders. Thus, rather than considering iteration limit to be an independent variable, we blocked on iteration limit level when performing our analyses. We did examine our data, however, to assess whether iteration limit effects existed.

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 increased (i.e. four increases per program, progressing from 5 to 10, 10 to 15, 15 to 20 and 20 to 25), coverage values for GDE increased as limits increased in all 16 cases, coverage values for GRE increased as limits increased in 14 of 16 cases, coverage values for GDN increased as limits increased in all 16 cases and coverage values for GRN increased as limits increased in all 16 cases. The coverage increases, however, were small overall—never more than two branches—and only 24 of 64 increases were statistically significant, which indicates that our genetic algorithm was converging.

Iteration trends occurred for the concolic algorithm as well; values generally increased by small amounts in all 64 cases. In this case, all of these increases were statistically significant, suggesting that iteration played 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 efficiency is concerned, iteration limits had larger effects. For the genetic algorithm, costs differed across iteration limits by relatively substantial amounts (i.e. by factors ranging from 4 to 6 from iteration limits 5 to 25). Where the concolic algorithm is concerned, we also saw increases in costs as iteration limits increased. The increases were smaller numerically than those observed with the genetic algorithm, but they were similar in terms of the factors involved (i.e. the increases ranged from 5 to 10 as iteration limits ranged from 1 to 9).

4.7.5. Initial test suite characteristics. Test suites can differ in terms of size, composition and coverage achieved. Such differences in test suite characteristics could potentially affect augmentation processes. For example, the extent to which an existing test suite achieves coverage prior to modifications can affect the number and locations of coverage elements that must be targeted by augmentation. Furthermore, test suite characteristics can impact the size and diversity of the starting populations utilized by test case generation techniques.

For these reasons, we chose to additionally examine our results in terms of four different fixed levels of coverage achieved by test suites. To do this, for each object program, we considered the total branch coverage achieved by the 100 test suites for that program, ranked the test suites in terms of coverage, and partitioned them into four equal size quartiles, denoted Q1, Q2, Q3 and Q4, respectively, where Q1 contains the 25 test suites achieving the lowest levels of coverage, Q2 contains the 25 suites achieving the next highest levels and so forth. We then conducted the same statistical tests on the resulting data that were conducted in examining our first and second research questions, on a per-quartile basis.

In three of the resulting comparisons, namely (1) the impact of test order on coverage, (2) the impact of test order on cost, and (3) the impact of test reuse on cost, we observed no differences in results across quartiles. That is, test suite characteristics did not impact the associated effects. In one of the resulting comparisons, however, namely (4) the impact of test reuse on coverage, we did observe effects.

Table XIV presents results relevant to this assessment. The table is similar to Table XII, but in this case, we provide separate results per program for each of the four quartiles in the rows labeled ‘Q1’, ‘Q2’, ‘Q3’ and ‘Q4’. Where Table XII revealed statistically significant coverage differences between approaches using existing test cases and approaches using existing plus new test cases in all but five cases, the per-quartile assessment exhibits many more cases in which differences are not statistically significant. This may be caused, in part, by the fact that these comparisons employ data sets that, being smaller, do not provide enough data to provide sufficient power to statistical tests. There does appear to be a tendency, however, for lower quartiles to exhibit significance more frequently than

DOI: 10.1002/stvr

Table XIV. Impact of test reuse in quartiles.

		Coverage			
		GDE versus GRN	GRE versus GRN	CDE versus CDN	CRE versus CRN
printtok1	Q1	N ***N N N N***	***N*** N ***N N N***	N ***N N N N***	N ***N N N N***
	Q2	***N*** N N N N	N N ***N N*** N	N N N N N	N N N N N
	Q3	N N ***N N*** N	N N N ***N*** N	N N N N N	N N N N N
	Q4	***N*** N N ***N*** E	N N N N E	= N N N N	= N N N N
printtok2	Q1	***N N*** N N ***N***	***N N*** N ***N N***	N ***N N N N***	N ***N N N N***
	Q2	***N*** N N N N	***N*** N N N N	N ***N N N N***	N ***N N N N***
	Q3	N N N N N	N N N N N	N = N = N	N = = = =
	Q4	N N E E E	N E E E E	N N = N N	N N N N N
replace	Q1	N N N N N	***N*** N ***N N N***	***N N N N N***	***N N N N N***
	Q2	N ***N*** N N ***N***	***N*** N N N N	N ***N N N N***	N ***N N N N***
	Q3	N N N N N	N N N N N	N ***N N N N***	N ***N N N N***
	Q4	N ***N*** N N N	***N*** N N ***N*** N	N N N ***N N***	= N ***N N N***
tcas	Q1	***N N N N N***	***N N N*** N ***N***	***N N N N N***	***N N N N N***
	Q2	***N*** N N N N	***N*** N N N N	N ***N N N N***	N N ***N N N***
	Q3	N ***N*** N N N	N N N N N	= N N = =	= = = N N
	Q4	N = N = =	N N N N N	= = = = =	= = = = =

Table XV. Results of concolic testing from scratch.

	printtok1		printtok2		replace		tcas	
Iteration level	Cost	Coverage	Cost	Coverage	Cost	Coverage	Cost	Coverage
1	1.4	111	0.5	87	1.5	56	0.2	71
3	3.4	111	0.9	92	2.5	78	—	—
5	5.3	111	1.3	101	3.8	78	—	—
7	7.8	111	1.9	110	6.2	78	—	—
9	9.6	111	2.3	115	8.8	78	—	—

higher quartiles. In other words, the coverage benefits of using new test cases in addition to existing ones may dissipate as the degree of coverage achieved by initial test suites increases. Further, on the most complex of the programs, `replace`, the efficacy of using new test cases dissipates more slowly for the concolic algorithm than for the genetic algorithm. This may indicate the potential for the algorithms to be differently influenced by initial test suite characteristics on programs of different characteristics, a suggestion that we return to in Section 6 following presentation of the results of our second study.

4.7.6. The benefits of augmentation. In Section 1, we conjectured that augmentation techniques working with existing test suites can perform better than augmentation techniques working without existing suites. To further consider this claim, we applied the concolic testing tool CREST from scratch on our programs, working without the benefit of test cases (the approach under which these algorithms have been traditionally been studied to date).[‡‡]

Table XV displays the results, listing the cost in seconds and the final coverage reached in branches on each program, per iteration level (leftmost column). Entries of the form '—' under `tcas` indicate cases where larger iteration limits are not needed. Comparing results with those for augmentation techniques reveals substantially poorer coverage on all programs but `tcas`, at costs that are relatively similar. The benefit of allowing the concolic approach to reuse test cases in the augmentation task is quite clear.

[‡‡]An equivalent investigation of the genetic algorithm is not possible, because that algorithm necessarily begins with existing test suites.

DOI: 10.1002/stvr

5. EMPIRICAL STUDY 2

The results of Study 1 suggest that target code element order and test case reuse approach can indeed have different impacts in the context of different augmentation techniques and that the two underlying test case generation techniques that we consider can indeed have different strengths on different programs. However, as we discussed in Section 4.5, the programs we used in that study are relatively small and simple. We wish to see whether the results of our first study generalize to larger, more complex programs. Thus, we replicated Study 1 on a considerably more complex open-source program, `grep`, for which a sequence of six versions was available.

For this study, we again consider the same research questions considered in Study 1, and for completeness, we repeat these here, designated as RQ1′, RQ2′ and RQ3′ in recognition of the different experimental contexts being utilized.

RQ1′: How does the order of consideration of target code elements affect augmentation techniques?
RQ2′: How does the manner of use of existing and newly generated test cases affect augmentation techniques?
RQ3′: How does the use of genetic and concolic test case generation techniques affect augmentation techniques?

As noted, this study utilizes the `grep` program provided in the Software-artifact Infrastructure Repository [75]. The `grep` program is a command-line text-search utility originally written for Unix. It searches files or standard input globally for lines matching a given regular expression and prints the lines to the program's standard output. It contains about 10 000 lines of C code. As mentioned earlier, `grep` is available with six sequential versions. However, the program does not have an enormous test universe of test cases offering complete coverage of the code; rather, it comes with a single test suite containing 792 test cases. We augment this test suite for each of the five versions after the base version. Table XVI provides details on the numbers of the branches for each of these subsequent versions, as well as the coverage achieved on each of those versions by the test suite prior to augmentation. (Note that in this study, as in our first study, we utilize the 'extended' versions of `grep` that are required by the concolic algorithm implementation.)

This study utilizes the same variables and measures as Study 1. It also possesses the same threats to validity as Study 1 with the exception of those specifically addressed in this study (size and representativeness of the object programs). We thus do not repeat discussion of these here. Instead, we describe only the differences between this study and Study 1. We then present data and analysis and discussion of results.

5.1. Experiment set-up

The `grep` program is quite different from the programs used in Study 1, in terms of its size and its initial test suite, and this required us to make some adjustments to the experiment process. First, one test case for `grep` has three parts: option, pattern and file. The option part includes command-line arguments that change many of the program's behaviours. For example, the option flag '-i' enables case-insensitive search. The pattern part is the regular expression that the user wishes to find in files. Therefore, both option and pattern parts are strings. The file part specifies where the user wishes to search for the pattern and is usually a path. We did not limit option and pattern lengths as we did for

Table XVI. Initial coverage information for `grep`.

Version	Total number of branches	Initial coverage
V1	3934	2151
V2	4146	2245
V3	4234	2271
V4	4262	2284
V5	4264	2284

DOI: 10.1002/stvr

the programs in our first study, as both lengths in the existing test cases are less than 30, which does not cause any problem for our test case generation techniques. For the file parameter, the existing test cases make use of five different files of which the largest contains 10 965 lines.

A second set of changes involves the settings used for the genetic algorithm, the first of which relate to the test suite reuse approach. With the genetic algorithm, if all test cases are used to form the initial population for a target, the test case generation process may take an inordinately long (and practically unreasonable) amount of time. In such cases, it is common to use a subset of the population [88]. To determine a reasonable subset size to use, we ran trials on the base version (V0) using initial sizes 25, 50, 100, 150 and 792. These trials covered 540, 613, 577, 634 and 623 branches separately in 5.6, 4.4, 4.7, 6.1 and 6.7 days, respectively. We determined that size 50 presented the best ratio of coverage to efficiency when applied to version V0. For a target, if there are more than 50 test cases reaching the method that contains it, we select the 50 fittest test cases as the initial population when we just use existing test cases. When we consider existing plus new test cases in the genetic technique, in addition to the 50 we chose, we add the newly generated test cases that reach the method containing the target into the initial population for the target. In this case, the existing plus new approach has more test cases to use for each target. In our experiment runs, we use that population size on subsequent versions. Note that this approach is practically reasonable in the context of evolving software, because engineers can tune a testing approach on an initial version and then use that tuned approach on subsequent versions.

We also altered the genetic algorithm process somewhat for use on `grep`. Every character in the option and pattern arguments to the program is treated as a gene in the chromosome. The whole file is also a gene—this is different from the approach used for the smaller programs but is necessary because the files used for `grep` are very large, and if we consider mutating the file content, it would be difficult for the concolic technique to do so. To be fair, however, in both techniques, we treat the file name itself as a manipulable input. For the genetic approach, this means that the file name is treated as a gene, and we can switch the file in the chromosome with other files in the file pool. We used the same strategies for fitness function, selection and crossover as on the smaller programs. We use a mutation rate of 0.05.

Finally, because the test case generation process takes much longer on `grep` than on the programs used in Study 1, rather than use five different iteration levels, we used just one. To make an informed decision as to an iteration level, we applied the following process to version V0. (Again, this is a process that engineers could apply on an initial version in practice in order to tune an approach for use on subsequent versions). We reasoned that an iteration level should be chosen based on the trade-off it presents with respect to costs and benefits. We used the following formula to examine these trade-offs:

$$\frac{(C(I_{k+3}) - C(I_k))/C(I_k)}{(T(I_{k+3}) - T(I_k))} \tag{1}$$

Here, $C(I_k)$ is the number of covered branches in the target program at the kth iteration level, and $T(I_k)$ is the execution time required to augment the test suite at the kth iteration level, measured in hours for the concolic algorithm and days for the genetic algorithm. The formula calculates the cost-benefit increase across the subsequent three versions to avoid local minima or maxima that may exist in calculating it across a single iteration level.

To choose an iteration level for the genetic and concolic algorithms, we applied each algorithm to version V0 of `grep` at increasingly higher iteration levels, applying the equation to each level as the data required for that level (from applications at subsequent levels) became available. We continued this process until the difference in ratios between two successive iterations fell below 0.01. In other words, after this point, it takes more than 1 h for the concolic approach and 24 h for the genetic approach to increase coverage by 1% when we run the experiment at the third higher level. This process ultimately led us to choose iteration level 11 for the concolic approach and iteration level 15 for the genetic approach.

DOI: 10.1002/stvr

Table XVII. Coverage and cost data for `grep`, per version and technique.

		Version											
		V1		V2		V3		V4		V5		Average	
Technique		D	R	D	R	D	R	D	R	D	R	D	R
						Coverage							
GA	E	584	575	557	592	607	590	594	636	656	593	599.6	597.2
	N	587	570	584	615	594	583	631	640	635	621	606.2	605.8
CT	E	390	390	405	405	423	423	448	448	448	448	422.8	422.8
	N	604	622	621	621	644	626	668	676	668	676	641.0	644.2
						Cost (h)							
GA	E	93.6	93.6	88.8	98.4	110.4	84.0	79.2	96.0	91.2	88.8	92.6	92.2
	N	160.80	163.2	146.4	184.8	208.8	182.4	132.0	163.2	180.0	213.6	165.6	181.4
CT	E	7.6	8.2	11.6	12.3	13.9	13.7	12.5	12.3	12.4	12.7	11.6	11.8
	N	28.3	28.2	40.4	41.1	46.3	43.2	34.9	39.7	35.7	39.9	37.1	38.4

Having selected the foregoing parameters, we proceeded with the experiment runs, in which we applied each augmentation technique to each of the five subsequent versions of `grep`. Because the algorithms do include non-deterministic behaviour, we applied each algorithm three times for each version. We thus obtained 60 data points on the program for each algorithm, in total (i.e. 2 test reuse approaches $*$ 2 target orders $*$ 5 versions $*$ 3 runs).

5.2. Results and analysis

Table XVII presents the data gathered for `grep`. The upper half of the table provides coverage data, and the lower half provides cost data. In each half of the table, the first two rows present the data for the genetic algorithm, and the last two rows present the data for the concolic algorithm. Coverage data are presented in terms of the numbers of previously uncovered branches (total number of branches − initial coverage in Table XVI) that the approach covered. Cost data are presented in hours. For each version and algorithm, four numbers are shown, corresponding to measurements gathered for the four combinations of target code element orders ('D' and 'R') and test case reuse approaches ('E' and 'N'). Each cell in the table shows the mean value across the three runs performed for the given combination.

Where coverage data are concerned, for the genetic algorithm, on average across all versions, using DFO and existing test cases covered 599.6 new branches while using existing plus new test cases added 606.2, just a 1.1% increase. Using random order, existing test cases covered 597.2 branches while existing plus new added 605.8, a 1.4% increase. Results varied across versions, however, with the use of existing plus new cases outperforming the use of just existing test cases on only three of five versions for each order (V1, V2 and V4 for DFO and V2, V4 and V5 for random). For DFO, the largest increase was 6.2% on V4, and the smallest was −3.2% on V5, while for random orders, the largest increase was 4.7% on V5 and the smallest was −1.2% on V3. Differences associated with target orders were also small on average (less than one branch), with no target order being predominantly better.

For the concolic algorithm, differences associated with test case reuse methods were greater. On average, across all versions, using DFO and existing test cases covered 422.8 new branches, while using DFO and existing plus new test cases covered 641.0, a 51.6% increase. Using random orders and existing test cases covered 422.8 new branches, while using random orders and existing plus new test cases covered 644.2 branches, a 52.4% increase. Improvements in results were consistent across versions and fell within relatively similar ranges, with the largest increase being 54.9% on V1 and the smallest being 49.1% on V4 and V5 for DFO and the largest increase being 59.5% on V1 and the smallest being 48% on V3 for random orders. Differences associated with target orders, however, continued to be small or none on average.

DOI: 10.1002/stvr

Where cost data are concerned, for the genetic algorithm, on average, across all versions, using DFO and existing test cases cost 92.6 h while using existing plus new test cases cost 165.6 h, a 78.8% increase. Using random order and existing test cases cost 92.2 h, while using random order and existing plus new test cases cost 181.4 h, a 96.9% increase. Results were consistent in direction across versions, varying in magnitude from 97.4% on V5 to 64.9% on V2 for DFO and from 117.1% on V3 to 70.0% on V4 for random orders. Differences between target orders, in contrast, were less consistent across versions. When using just existing test cases, there was no average difference (and no clear winner) between DFO and random orders. When using existing plus new test cases, there was a 15.84 h average difference favouring DFO, with DFO outperforming random on all but V3.

For concolic testing, on average, across all versions, using DFO and existing test cases cost 11.6 h, while using DFO and existing plus new test cases cost 37.1 h, a 220.0% increase. Using random order and existing test cases cost 11.8 h while using random order and existing plus new test cases cost 38.4 h, a 224.5% increase. Results were again consistent in direction across versions, varying in magnitude from 272.4% on V1 to 179.2% on V4 for DFO and from 243.9% on V1 to 214.2% on V5 for random. Differences between test case orders, however, were inconsistent across versions and relatively small on average (e.g. 1.3 h when using existing plus new test cases and 0.2 h when using just existing test cases).

Finally, where comparisons of the test case generation algorithms are concerned, when using just existing test cases, the genetic algorithm attained substantially higher coverage (from 37.5% to 49.7%) across the five versions than the concolic algorithm. When using existing plus new test cases, however, the concolic algorithm outperformed the genetic algorithm, from amounts ranging from 1.0% to 9.1% across versions. Also, in all cases, the concolic algorithm was substantially faster than the genetic algorithm.

5.3. Discussion and implications

We begin by summarizing the results for `grep`, as follows:

- DFO and random orders had little effect on coverage differences, for both the genetic and concolic approaches and under both test case reuse approaches.
- DFO and random orders had inconsistent and varying effects on the costs of genetic and concolic approaches in general. The one combination of treatments in which order had an impact occurred when using both existing and new test cases with the genetic algorithm.
- The concolic algorithm benefitted substantially in terms of coverage when using existing plus new test cases rather than just existing test cases, and this benefit occurred for both test case orders. The genetic algorithm benefitted only mildly and less consistently.
- In all cases, using existing plus new test cases added substantial costs to the test case generation process.
- The concolic approach outperformed the genetic approach in terms of coverage when using existing plus new test cases, while the genetic approach was better when using just existing test cases.

The foregoing results are similar in their overall trends to those seen in Study 1, with the exception of the last. We believe that the differences observed for the concolic approach are primarily due to the fact that initial test suites achieved much lower levels of coverage on `grep` than did the initial test suites used in Study 1; thus, new test cases that are generated had greater potential to lead to additional coverage simply because more targets were available. The fact that the genetic algorithm did not achieve a similar level of improvement, on the other hand, is likely due to the fact that the newly generated test cases did not provide better power than the existing test cases, which is consistent with what we observed on the smaller programs.

The data also prompt additional observations. On this much larger program, the costs associated with the test generation task were much greater than on the smaller programs. There was still a cost-benefit trade-off involved, for both techniques, in choosing to use existing or existing plus new test cases, but the benefit-to-cost ratio for the genetic algorithm was much smaller here than with the first four programs, and the benefit-to-cost ratio for the concolic algorithm was much larger. Thus,

DOI: 10.1002/stvr

the cases in which using existing plus new test cases would be worthwhile are likely to occur much less often for the genetic algorithm than for the concolic algorithm.

6. ADDITIONAL ANALYSIS AND IMPLICATIONS

Our two studies revealed overall performance differences between augmentation techniques utilizing different test case generation algorithms and suggested several reasons for those differences. To obtain further insights, we analysed the differences in coverage results between the techniques in greater detail, and we present the results of that analysis here. The results also prompt us to explore the possibility of a hybrid augmentation technique that combines genetic and concolic test case generation approaches; we discuss several factors that may be important to consider in creating such a technique.

6.1. Overall comparison

We begin by considering results of Study 1. Table XVIII shows the differences in branch coverage achieved by concolic and genetic test case generation techniques in that study. The table shows, for each of the four programs, for each of the four techniques applied and for the iteration level at which the most effective results were seen, the average numbers of branches across 100 test suites such that (1) (GA–CT) test cases generated by the genetic algorithm covered that branch while no test cases generated by the concolic algorithm covered it; (2) (CT–GA) test cases generated by the concolic algorithm covered that branch while no test cases generated by the genetic algorithm covered it; (3) (GA ∩ CT) each algorithm succeeded in generating at least one test suite that covered the branch; (4) (GA ∪ CT) one or both algorithms succeeded in generating at least one test suite that covered the branch. As the table shows, for all techniques and programs, each of the two algorithms (concolic and genetic) was able to cover at least some branches that could not be covered by the other algorithm. On `tcas`, the smallest of the four programs, the numbers are small (between 0.01 and 1.39 branches). On the other three programs, larger ranges of branch coverage differences occurred, with the genetic algorithm accounting for more differences on `printtok1` and `printtok2` and the concolic algorithm accounting for more on `replace`.

We provide further details on two of the programs in Figure 2. The figure focuses on the two object programs on which the techniques exhibited the greatest range of differences, `replace` and `printtok1`, and on the case in which DFO and new plus existing test cases are utilized. For each of these two cases, the figure displays a graph. The x-axes in these graphs correspond to branches (branch identifier numbers) in the program. The y-axes indicate the numbers of test suites (from among the 100 suites used) in which each branch was covered, with the bar extending upward from the line labeled '0' showing results for the concolic algorithm and the bar extending downward from that line showing coverage for the genetic algorithm.

In the case of `replace`, we see that a relatively small number of branches (13 to be precise) were not covered by any of the 100 test suites, for either technique. A much larger number (101 to

Table XVIII. Branch coverage differences—smaller programs.

Program	GA–CT	CT–GA	GA ∩ CT	GA∪ CT	GA–CT	CT–GA	GA ∩ CT	GA∪ CT
		DFO/EXISTING				DFO/NEW		
`printtok1`	5.25	0.27	152.23	157.75	5.65	0.11	152.76	158.52
`printtok2`	4.39	1.17	171.90	177.54	4.02	1.13	172.50	177.71
`replace`	4.23	5.70	183.90	193.84	3.35	5.29	185.46	194.09
`tcas`	1.37	0.10	69.44	70.91	0.13	0.01	70.86	71.00
		RAND/EXISTING				RAND/NEW		
`printtok1`	5.32	0.15	152.35	157.82	5.21	0.13	152.64	157.98
`printtok2`	4.35	1.14	172.02	177.51	4.02	1.08	172.63	177.7
`replace`	4.28	5.67	183.93	193.88	3.28	5.22	185.58	194.08
`tcas`	1.39	0.04	69.50	70.93	0.13	0.02	70.80	71.00

DOI: 10.1002/stvr

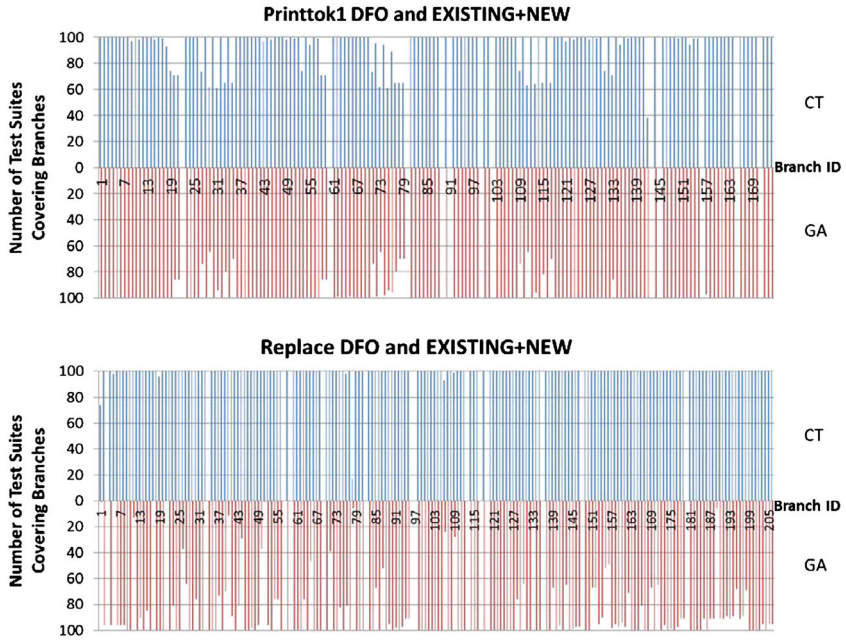

Figure 2. Comparison of branch coverage behaviours for concolic and genetic algorithms on two representative cases.

Table XIX. Branch coverage differences—`grep`.

Version	GA–CT	CT–GA	GA ∩ CT	GA∪ CT	GA–CT	CT–GA	GA ∩ CT	GA ∪ CT
		DFO/EXISTING				DFO/NEW		
V1	541	236	154	931	375	302	302	979
V2	553	293	112	958	403	334	287	1024
V3	531	270	153	954	383	349	295	1027
V4	546	293	155	994	394	341	327	1062
V5	565	285	163	1013	382	334	334	1050
		RAND/EXISTING				RAND/NEW		
V1	521	243	147	911	342	323	299	964
V2	559	278	127	964	411	355	266	1032
V3	559	272	151	982	398	356	270	1024
V4	548	276	172	996	374	319	357	1050
V5	518	292	156	966	369	343	333	1045

be precise) were covered by all 100 test suites, for both techniques. The remaining branches were missed for at least some test suites by one or both algorithms. For the concolic algorithm, only a few such branches (seven to be precise) were missed by between 1 and 99 test suites, while for the genetic algorithm, far more (86 to be precise) were missed by between 1 and 99 test suites. In other words, the concolic algorithm achieved much higher rates of success in covering branches than the genetic algorithm on a large number of branches.

The `printtok1` object yields a different picture. Here again, several branches were left uncovered by both techniques, but the genetic technique was 100% successful on a few more branches (22 to be precise) than the concolic approach, and the genetic approach had somewhat higher success at covering branches that are not always covered. The differences between the two algorithms on this object program, however, were not as large as those seen on `replace`.

We next turn our attention to Study 2 and `grep`. In this case, because the executions per version involve independent runs of techniques, we cannot compare differences per run; instead, we choose a different approach. Table XIX shows, for each of the five versions of `grep` and for each of the

DOI: 10.1002/stvr

Table XX. Numbers of times in which branches in `grep` were covered by one, two or three test suites, for depth-first order with existing and new test cases.

	GA only			CT only		
Version	1	2	3	1	2	3
V1	25	11	339	0	0	302
V2	13	16	374	0	0	334
V3	16	11	356	0	0	349
V4	13	21	360	0	0	341
V5	6	10	366	0	0	334

four techniques applied, the numbers of branches such that (1) (GA–CT) at least one of the three test suites generated by the genetic algorithm covered that branch while no test suites generated by the concolic algorithm covered it; (2) (CT–GA) at least one of the three test suites generated by the concolic algorithm covered that branch while no test suites generated by the genetic algorithm covered it; (3) (GA $\cap$ CT) each algorithm succeeded in generating at least one test suite that covered the branch; and (4) (GA $\cup$ CT) one or both algorithms succeeded in generating at least one test suite that covered the branch.

As the table shows, on the larger `grep` object, the genetic and concolic algorithms exhibited large disparities in their abilities to cover specific branches. For example, for the scenario in which DFO and existing test cases only were used, both algorithms jointly were able to cover between 112 and 163 branches across the five versions, but the numbers of branches covered only by the concolic algorithm exceeded these numbers by factors of between 0.5 and 0.8, and the number of branches covered only by the genetic algorithm exceeded these numbers by a factor of between 2.5 and 3.9. Similar trends (although with different increase factors) can be seen in the other scenarios. Clearly, in this more complex program, the differences in coverage abilities of the two algorithms were larger than those seen on the smaller, less complex programs.

Table XX considers these differences further for the case in which DFO and existing plus new test cases are used. For each version of `grep`, the table displays data about just those branches that are covered only by the genetic test case generation or only by the concolic test case generation. The data denote the numbers of times these branches were covered by only one of the test suites generated, only two of the test suites generated or all three of the test suites generated. The table shows a trend observed generally (across all four augmentation techniques) on the program: the concolic approach either succeeded or failed in all cases (on all test suites created), whereas the genetic algorithm often encountered branches that are covered only probabilistically, that is, on some test suites generated but not on others.

6.2. Analysis of specific branches

To further understand the differences in technique performance, we selected several branches from `replace`, `printtok1` and `grep` on which such differences occurred and analysed them to determine causes of the differences. On `replace`, we selected the seven branches that exhibited the most extreme differences in results in cases where the concolic algorithm greatly outperformed the genetic algorithm. On `printtok1`, we selected the seven branches that exhibited the most extreme differences in results in cases where the genetic algorithm greatly outperformed the concolic algorithm. On `grep`, where we have only three test suites, we could not locate branches that were outliers, so instead, we randomly sampled four branches that were easy for the concolic approach to cover but not for the genetic approach to cover and four branches in which this situation was reversed.

Considering `replace` first, we were able to classify the seven branches on which the concolic algorithm outperformed the genetic algorithm into three groups based on three overall observed causes of problems in coverage.

DOI: 10.1002/stvr

The first group (G1) of branches relates to limitations in the mutation pool settings chosen for the genetic algorithm. One of the seven branches falls into this group. In `replace`, there is a predicate that checks the number of input arguments provided to the program, and the program needs to be given fewer than two arguments to cover the 'true' branch out of this predicate. In the initial population of test cases provided to the algorithm, however, all test cases have two or three arguments, and we did not include the choice of mutating the number of inputs as part of our mutation pool. Thus, the genetic algorithm can never cover the branch. To cover this branch with the genetic approach, we would need to have sufficient knowledge of the program internals to cause us to change this behaviour, perhaps via a pre-processing static analysis. In our study, we treated the programs as black boxes for the genetic algorithm, and tuning was performed based on program specifications, inputs and environment conditions. In contrast, the concolic approach treats program as white boxes, and applying it requires testers to consider program internals. Thus, for the concolic approach, we specified the number of arguments as a symbolic value, and this allowed us to cover the branch in question on every run.

The second group (G2) of branches also involves mutation pool settings, but of a different type, and three of the seven branches belong to it. There are several branches in `replace` such that, for those branches to be taken, characters in specific strings must equal the NULL character. Because we did not include this character in our mutation pool, the only way in which it would occur in a test case would be if it occurred in the initial test case population, and this is infrequent. Thus, it is difficult for the genetic algorithm to cover such branches. Including all possible characters in the mutation pool could remedy this but would increase the search space and cost of the approach substantially. Further analysis of the program could also remedy this, at the cost of such analysis. In contrast, the concolic approach did not exclude the character.

The third group (G3) of branches involves the presence of deeply nested `if` branches, and three branches belong to it. Predicates in deeply nested branches pose a well-known problem for genetic algorithms, although the algorithms can be helped through specific program transformations [89]. For example, in `replace`, there is one branch in a function named `in_set_2`. This is in the first `if` statement in that function, but this function is called at the 10th level of its callee function `makepat`. Above `makepat`, there are two other functions. To cover this branch, a test case must satisfy several conditions. The genetic algorithm had no 'knowledge' of these conditions and simply attempted to proceed in a general search direction; thus, it was difficult for the algorithm to satisfy all the conditions at once. Here too, the concolic approach, by design, had no problem.

Considering `printtok1`, we were able to classify the seven branches considered into two groups. The first group (G4) contains one branch, and the failure of the concolic algorithm to cover it was related to limitations of CREST involving pointer arithmetic and nonlinear arithmetic. More specifically, `printtok1` contains a predicate `check_delimiter()` that contains the `isalpha()` and `isdigit()` C standard macro functions. Both of these functions use the bitwise `&` operator and pointer arithmetic. To cover this branch using concolic testing, we would need to use an implementation that supports bitwise operators by employing bit-vector logic and handle pointer arithmetic by providing a memory model. In contrast, the genetic approach was not affected by complex expressions such as this because it does not attempt to solve path constraints.

The failure of the concolic approach to cover the second group (G5) of branches, including the other six, was due to iteration limits. The `printtok1` program includes a `next_state()` function that uses a symbolic input character as an index into an array of characters. Because CREST does not support accesses to array elements through a symbolic index variable, it transforms the process to use `if-then-else` statements to handle all possible values of the symbolic index variable one by one. For example, consider a program containing the following function:

```
01:void f(int x){
02: if (x == 10){ ... }
03: else if (x == 20){ ... }
04: else if (x == 30){ ... }
05: ...  }
```


DOI: 10.1002/stvr

Given the foregoing, for a symbolic `unsigned char` variable `i`, the code `int next_state(int i) { ... if(a[i]==C) f(b[i]);...}` is transformed into the following code where `a` is an array of characters, `b` is an array of integers (suppose that `b[i]=i+10`; i.e. `b[0]=10`, `b[1]=11, ...`) and `C` is a character constant:

```
06:int next_state(int i){...
07: // Transformation of
08: // if(a[i]==C) f(b[i]);
09: if(i==0 && a[0]==C)          f(b[0]);
10: else if(i==1 && a[1]==C)     f(b[1]);
11:  ...
12: else if(i==255 && a[255]==C)f(b[255]);
13: ... }
```

However, this transformation still does not solve the problem completely. Suppose that the concolic approach tries to cover the branches in `f()` (lines 2–4). The concolic approach controls the symbolic variable `i` that is passed to `next_state()` as a parameter (line 6) and controls the parameter to `f()` indirectly (lines 9–12). In other words, to cover the branches in `f()`, the concolic approach must try corresponding different branches in `next_state()` (i.e. a maximum of 256 different values for symbolic variable `i`). Given an iteration limit less than 10, there is little chance for the approach to reach all branches in `f()`. For example, suppose that a target branch b_t is the `then` branch of `f()` at line 2 (i.e. `x==10`). Also suppose that an initial value of `i` is 255, which makes the first symbolic execution path be $\neg(i = 0 \wedge a[0] = C) \wedge \neg(i = 1 \wedge a[1] = C) \ldots \wedge \neg(i = 254 \wedge a[254] = C) \wedge (i = 255 \wedge a[255] = C) \wedge \neg(x = 10) \ldots$ (see the rightmost execution path in Figure 3). To cover b_t, Algorithm 3 has to iterate through lines 8–16 255 more times, as b_t can be covered by only the leftmost execution path in Figure 3. However, this is not possible because $n_{iter} < 10$ in our experiments (see line 8 in Algorithm 3). In contrast, the genetic approach may reach any of the branches if it succeeds in choosing appropriate inputs.

Finally, we turn to `grep`. Of the four branches on which the genetic approach had difficulties, one (group G6) was a deeply nested branch, similar to the case discussed earlier with respect to `replace`. The other three branches (group G7) are all incident on `malloc` attempts and taken when that routine fails because of the exhaustion of memory. It is virtually impossible for the genetic approach to generate test cases for `grep` that consume enough memory to trigger coverage of these branches. The concolic approach, however, covered them, but this is actually a side effect rather than a direct effect. This is because the concolic algorithm saves execution path information for test cases, and eventually, this path information can consume enough memory to cause `malloc` failures.

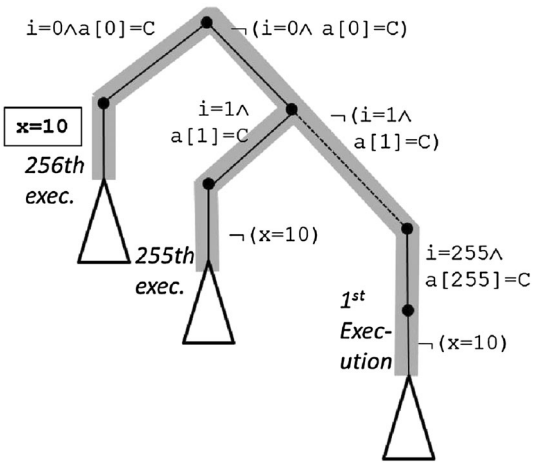

Figure 3. Symbolic execution tree of the example code.

DOI: 10.1002/stvr

For the four branches on which the concolic approach had difficulties, we identified two groups, each relevant to two of the branches. The first group (G8) is related to external binary library functions such as `strcmp()` and `strlen()`. Branches belonging to this group are taken based on results of these binary library functions. These functions cannot be analysed by the concolic algorithm, and thus, it failed to generate test cases that cover them. The genetic approach does not need to analyse the functions and did select inputs that cover the branches.

The second group (G9) of branches are related to dynamic memory management. For example, `grep` transforms a given regular expression pattern into a deterministic finite automaton (DFA) and stores the DFA in a buffer. Before `grep` stores the DFA into the buffer, it should check whether the size of the buffer is large enough to contain the DFA. If not, `grep` extends the buffer. Because the concolic approach cannot control the size of the DFA directly via path conditions, it is difficult for it to cover branches that compare the size of the buffer and the size of the DFA. The genetic approach, however, because of the diversity created through crossover and mutation, can by chance end up with test cases that vary the DFA size as needed.

Table XXI summarizes the foregoing results. For each of the groups identified, the table lists the program(s) that group occurred in, the number of branches, the algorithm that exhibited the weakness in achieving coverage, and the cause of the weakness. The rightmost column in the table classifies the observed weaknesses into three categories, as follows.

The first broad category of weaknesses (groups G1 and G2, four branches) involved tuning limitations (mutation pool settings) and occurred only for the genetic algorithm. Such weaknesses will necessarily occur for that algorithm because of the way in which the algorithm must be applied; however, in practice, they could be partly addressed by tuning the algorithm better, which is particularly possible in the context of an evolving program as test suites are reused and improved on subsequent versions.

The second broad category of weaknesses (group G4, one branch) involved effects related to implementations and occurred only for the concolic algorithm. In this case, the failure of the technique is not algorithmic but rather is due to the specific implementation of the algorithm and could be addressed through improvements in implementations. For example, the concolic approach *could* be implemented to better handle nonlinear arithmetic.

The third broad category of weaknesses (groups G3, G5, G6, G7, G8 and G9, 18 branches) involves neither tuning problems nor implementation problems but rather lies in the natures of the algorithms themselves. Genetic algorithms are simply not likely to handle deeply nested ifs (groups G3 and G6), whereas concolic algorithms can. Concolic algorithms are simply not able to handle non-analysable external libraries or dynamic memory management issues (groups G8 and G9). We also place group G5 in this category. While we selected iteration limits, and thus, they might be seen as a matter of tuning, at the core of the concolic approach, some limit will be needed as an algorithmic matter, and there could exist programs such that, for any limit selected, that limit is not sufficient to allow certain branches to be reached. Finally, regarding group G7, the fact that the concolic implementation could cover branches incident on malloc failures is related to the algorithm's need to collect data that can exceed available memory.

Table XXI. Summary of coverage limitations.

Group	Program	Number of branches	Weak algorithm	Specific cause	Classification
G1	`replace`	1	GA	Limitations in mutation pool setting (arguments)	Tuning
G2	`replace`	3	GA	Limitations in mutation pool setting (NULL char)	Tuning
G3	`replace`	3	GA	Deeply nested ifs not reached	Algorithmic
G4	`printtok1`	1	CT	Limitations handling arithmetic constructs	Implementation
G5	`printtok1`	6	CT	Iteration limits and loops	Algorithmic
G6	`grep`	1	GA	Deeply nested ifs not reached	Algorithmic
G7	`grep`	3	GA	Malloc failures not covered	Algorithmic
G8	`grep`	2	CT	External libraries not analysable	Algorithmic
G9	`grep`	2	CT	Dynamic memory management not controlled	Algorithmic

DOI: 10.1002/stvr

6.3. Towards a hybrid algorithm

While approaches for combining different test case generation techniques have been studied (as discussed in Section 2.2), these approaches have not yet been investigated in the context of test suite augmentation. The differences observed between the concolic and genetic algorithms in our data and additional analysis suggest that augmentation techniques that combine both approaches should be more cost-effective than approaches that utilize just single techniques.[§§] Such hybrid algorithms could enable an overall test case generation approach that addresses the algorithmic limitations seen in the approaches singly.

The results of our study suggest that such a hybrid approach should be structured as follows.

- The concolic approach was much more efficient than the genetic approach, so a hybrid approach should begin with concolic testing first and let it cover as many branches as possible before passing control to the genetic algorithm.
- Given that the test reuse approach had an impact on effectiveness for concolic testing only, when we use the concolic algorithm, we should add new test cases as the process goes on. In contrast, test reuse had little impact on the coverage for the genetic algorithm while significantly increasing its cost. Thus, when we use the genetic algorithm, we can rely on existing test cases in our initial population for every target.
- Order of targets can impact the efficiency of the genetic algorithm but had no effect on the concolic algorithm. Thus, we can order the targets for the former but need not do so for the latter.
- Figure 2 and Table XX illustrate that the concolic testing algorithm was usually consistent across runs (i.e. it tended to cover the same branches in each run using the same existing test cases), while the genetic approach was far more non-deterministic. Therefore, it should suffice to execute the concolic approach just once if there is no change in existing test cases, while we may benefit from running the genetic algorithm multiple times even when existing test cases do not change.

7. CONCLUSIONS AND FUTURE WORK

In this work, we have focused on test suite augmentation utilizing genetic and concolic algorithms, and on factors that affect its cost-effectiveness. The results of our studies show that the primary factor affecting augmentation was the test case generation algorithm utilized; this affected both cost and effectiveness. The manner in which existing and newly generated test cases were utilized also had a substantial effect on efficiency and in some cases a substantial effect on effectiveness. The order in which target code elements were considered turned out to have relatively few effects when using concolic test case generation but in some cases influenced the efficiency of genetic test case generation. The results of our first study, on four relatively small programs using a large number of test suites, were supported by our second study of a much larger program available in multiple versions. Together, the studies reveal a potential opportunity for creating a more cost-effective hybrid augmentation approach leveraging both concolic and genetic test case generation techniques, while appropriately utilizing our understanding of the factors that affect them.

Our results have several implications for the creation and further study of augmentation techniques. Perhaps the most intriguing result stems from the observed complementariness of the concolic and genetic test case generation approaches, and the implications this raises for the prospects of hybrid approaches. 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.

[§§] Of course, such a hybrid approach can function only in cases in which both concolic and genetic test case generation are possible, and each class of approaches, as just discussed, has limitations to applicability.

DOI: 10.1002/stvr

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 3.2.2 and 3.2.3. Characteristics of program modifications also matter. Finally, algorithms used to identify target code differ and may affect the relative cost-effectiveness of subsequent augmentation algorithms. More formal studies of these factors could be helpful.

In closing, the results of this research provide many insights into the test suite augmentation problem, techniques for addressing it and factors that affect those techniques. Given all of the factors involved in the problem, however, including those not yet studied, a great deal of further research is needed. Through such research, we hope to be able to make cost-effective, automated techniques for test suite augmentation available to software engineers.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under awards CNS-0454203 and CCF-1161767 and by the Air Force Office of Scientific Research through award FA9550-10-1-0406 to the University of Nebraska–Lincoln. Also, this work was supported in part by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT and Future Planning (MSIP)/National Research Foundation (NRF) of Korea (grant NRF-2008-0062609), the Information Technology Research Center support program funded by MSIP and supervised by the National IT Industry Promotion Agency, Korea (NIPA-2014-H0301-14-1023), the NRF Mid-career Researcher Program funded by MSIP, Korea (NRF-2012R1A2A2A01046172) and the IT R&D Program of Ministry of Knowledge Economy/Korea Evaluation Institute of Industrial Technology (10041752). We thank Yuyang Liu for her valuable input on concolic testing of the object programs.

REFERENCES

1. Leung HKN, White L. Insights into regression testing. *Proceedings of the International Conference on Software Maintenance,* Miami, FL, 1989; 60–69.
2. Onoma K, Tsai W-T, Poonawala M, Suganuma H. Regression testing in an industrial environment. *Communications of the ACM* 1998; **41**(5):81–86.
3. Rothermel G, Harrold MJ. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 1997; **6**(2):173–210.
4. Apiwattanapong T, Santelices R, Chittimalli PK, Orso A, Harrold MJ. Matrix: maintenance-oriented testing requirements identifier and examiner. *Testing: Academic and Industry Conference on Practical Research and Techniques,* Windsor, UK, 2006; 137–146.
5. Person S, Dwyer MB, Elbaum S, Păsăreanu CS. Differential symbolic execution. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering,* Atlanta, GA, USA, November 2008; 226–237.
6. Santelices R, Chittimalli PK, Apiwattanapong T, Orso A, Harrold MJ. Test-suite augmentation for evolving software. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering,* L'AQUILA, ITALY, 2008; 218–227.
7. Xu Z, Rothermel G. Directed test suite augmentation. *Proceedings of the Asia-Pacific Software Engineering Conference,* Penang, Malaysia, 2009; 406–413.
8. Binkley D. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering* 1997; **23**(8):498–516.
9. Gupta R, Harrold MJ, Soffa M. Program slicing-based regression testing techniques. *Journal of Software Testing, Verification, and Reliability* June 1996; **6**(2):83–111.
10. Rothermel G, Harrold MJ. Selecting tests and identifying test coverage requirements for modified software. *Proceedings of the International Symposium on Software Testing and Analysis,* Seattle, WA, USA, 1994; 169–184.
11. Díaz E, Tuya J, Blanco R, Javier Dolado José. A tabu search algorithm for structural software testing. *Journal of Computers and Operations Research* 2008; **35**(10):3052–3072.
12. Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering,* Lisbon, Portugal, 2005; 263–272.
13. Waeselynck H, Thévenod-Fosse P, Abdellatif-Kaddour O. Simulated annealing applied to test generation: landscape characterization and stopping criteria. *Empirical Software Engineering: An International Journal* 2007; **12**(1):35–63.
14. Ferguson R, Korel B. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(1):63–86.
15. Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation,* Chicago, Illinois, USA, 2005; 213–223.
16. Xu Z, Cohen M, Rothermel G. Factors affecting the use of genetic algorithms in test suite augmentation. *Proceedings of the Genetic and Evolutionary Computation Conference,* Portland, OR, USA, 2010; 1365–1372.

DOI: 10.1002/stvr

17. Xu Z, Kim Y, Kim M, Rothermel G, Cohen MB. Directed test suite augmentation: techniques and tradeoffs. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering,* Santa Fe, New Mexico, November 2010; 257-266.
18. Chen YF, Rosenblum DS, Vo KP. TestTube: a system for selective regression testing. *Proceedings of the International Conference on Software Engineering,* Sorrento, Italy, 1994; 211–220.
19. Orso A, Shi N, Harrold MJ. Scaling regression testing to large software systems. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering,* Newport Beach, Ca, Usa, 2004; 241–251.
20. Ren X, Shah F, Tip F, Ryder BG, Chesley O. Chianti: a tool for change impact analysis of Java programs. *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages, and Applications,* Vancouver, British Columbia, Canada, October 2004; 432–448.
21. Ruth M, Oh S, Loup A, Horton B, Gallet O, Mata M, Tu S. Towards automatic regression test selection for web services. *Proceedings of the International Computer Software and Applications Conference,* Beijing, China, 2007; 729–736.
22. Yoo S, Harman M. Pareto efficient multi-objective test case selection. *Proceedings of the International Symposium on Software Testing and Analysis,* London, United Kingdom, 2007; 140–150.
23. Elbaum S, Malishevsky AG, Rothermel G. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering* 2002; **28**(2):159–182.
24. Li Z, Harman M, Hierons R. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering* 2007; **33**(4):225–237.
25. Walcott A, Soffa ML, Kapfhammer GM, Roos RS. Time-aware test suite prioritization. *Proceedings of the International Symposium on Software Testing and Analysis,* Portland, ME, USA, 2006; 1–12.
26. Bohner S, Arnold R. *Software Change Impact Analysis*. IEEE Computer Society Press: Los Alamitos, CA, 1996.
27. Person S, Yang G, Rungta N, Khurshid S. Directed incremental symbolic execution. *Programming Language Design and Implementation,* San Jose, CA, USA, 2011; 504–515.
28. Taneja K, Xie T, Tillmann N, Halleux J, Schulte W. eXpress: guided path exploration for regression test generation. *Proceedings of the International Symposium on Software Testing and Analysis,* Toronto, ON, Canada, 2011; 1–11.
29. Santelices R, Harrold MJ. Applying aggressive propagation-based strategies for testing changes. *International Conference on Software Testing, Verification, and Validation,* Berlin, German, 2011; 11–20.
30. Qi D, Roychoudhury A, Liang Z. Test generation to expose changes in evolving programs. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering,* Antwerp, Belgium, 2010; 397–406.
31. Jin W, Orso A, Xie T. Automated behavioral regression testing. *International Conference on Software Testing, Verification, and Validation,* Paris, France, 2010; 137–146.
32. Yoo S, Harman M. Test data augmentation: generating new test data from existing test data. *Technical Report TR-08-04,* King's College London, 2008.
33. Chang J, Richardson D. Structural specification-based testing: automated support and experimental evaluation. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering,* Toulouse, France, 1999; 285–302.
34. Marinov D, Khurshid S. TestEra: a novel framework for automated testing of Java programs. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering,* San Diego, USA, 2001; 22–31.
35. Offutt J, Abdurazik A. Generating tests from UML specifications. *Proceedings of the International Conference on UML,* Fort Collins, CO, USA, 1999; 416–429.
36. Avritzer A, Weyuker EJ. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering* 1995; **21**(9):705–716.
37. Hartman A, Nagin K. Model driven testing—AGEDIS architecture interfaces and tools. *Proceedings of the European Conference on Model Driven Software Engineering,* Nuremberg, Germany, 2003; 1–11.
38. Visser W, Pasareanu C, Khurshid S. Test input generation with Java Pathfinder. *Proceedings of the International Symposium on Software Testing and Analysis,* Boston, MA, USA, 2004; 97–107.
39. Briand LC, Labiche Y, He S. Automating regression test selection based on UML designs. *Information and Software Technology* 2009; **51**:16–30.
40. Pilskalns O, Uyan G, Andrews A. Regression testing UML designs. *Proceedings of the International Conference on Software Maintenance,* Philadelphia, Pennsylvania, USA, 2006; 254–264.
41. Bird D, Munoz C. Automatic generation of random self-checking test cases. *IBM Systems Journal* 1983; **22**(3): 229–245.
42. Chen TY, Merkel R. Quasi-random testing. *IEEE Transactions on Reliability* 2007; **56**(3):562–568.
43. Clarke L. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering* 1976; **2**(3):215–222.
44. DeMillo R, Offutt A. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 1991; **17**(9):900–910.
45. Gotlieb A, Botella B, Reuher M. Automatic test data generation using constraint solving techniques. *Proceedings of the International Symposium on Software Testing and Analysis,* Clearwater Beach, FL, USA, 1998; 53–62.
46. Korel B. Automated software test data generation. *IEEE Transactions on Software Engineering* 1990; **16**(8):870–897.
47. Baresel A, Binkley D, Harman M, Korel B. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. *Proceedings of the International Symposium on Software Testing and Analysis,* Boston, MA, USA, 2004; 108–118.

DOI: 10.1002/stvr

48. Michael C, McGraw G, Shatz M. Generating software test data by evolution. *IEEE Transactions on Software Engineering* 2001; **27**(12):1085–1110.
49. Pargas RP, Harrold MJ, Peck RR. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification, and Reliability* 1999; **9**:263–282.
50. Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. Exe: automatically generating inputs of death. *Proceedings of the Conference on Computing Communications and Security,* Alexandria, VA, USA, 2006; 322–335.
51. Clarke L, Richardson D. Applications of symbolic evaluation. *Journal of Systems and Software* 1985; **5**(1):15–35.
52. Sen K, Agha G. JCUTE: concolic unit testing and explicit path model-checking tools. *Proceedings of the International Conference on Computer Aided Verification,* Seattle, WA, USA, 2006; 419–423.
53. Emmi M, Majumdar R, Sen K. Dynamic test input generation for database applications. *Proceedings of the International Symposium on Software Testing and Analysis,* London, United Kingdom, 2007; 151–162.
54. Artzi S, Kiezun A, Dolby J, Tip F, Dig D, Paradkar A, Ernst MD. Finding bugs in dynamic web applications. *Proceedings of the International Symposium on Software Testing and Analysis,* Seattle, WA, USA, 2008; 261–272.
55. Wassermann G, Yu D, Chander A, Dhurjati D, Inamura H, Su Z. Dynamic test input generation for web applications. *Proceedings of the International Symposium on Software Testing and Analysis,* Seattle, WA, USA, 2008; 249–260.
56. Available from: http://babelfish.arc.nasa.gov/trac/jpf [last accessed 29 October 2014].
57. CREST—automatic test generation tool for C. Available from: http://jburnim.github.io/crest/ [last accessed 29 October 2014].
58. Cadar C, Dunbar D, Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation,* San Diego, CA, 2008; 209–224.
59. Tillmann N, Schulte W. Parameterized unit tests. *Proceedings of the ACM Symposium on Foundations of Software Engineering,* Lisbon, Portugal, 2005; 253–262.
60. Lakhotia K, Harman M, Gross H. AUSTIN: a tool for search based software testing for the C language and its evaluation on deployed automotive systems. *Proceedings of the international Symposium on Search Based Software Engineering,* Benevento, Italy, 2010; 101–110.
61. Fraser G, Arcuri A. EvoSuite: automatic test suite generation for object-oriented software. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering,* Szeged, Hungary, 2011; 416–419.
62. Inkumsah K, Xie T. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering,* Atlanta, Georgia, 2007; 297–306.
63. Borges M, d'Amorim M, Anand S, Bushnell D, Pasareanu C. Symbolic execution with interval solving and meta-heuristic search. *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation,* Montreal, QC, Canada, 2012; 111–120.
64. Baars A, Harman M, Hassoun Y, Lakhotia K, McMinn P, Tonella P, Vos T. Symbolic search-based testing. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering,* Oread, Lawrence, Kan, 2011; 53–62.
65. Malburg J, Fraser G. Combining search-based and constraint-based testing. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering,* Oread, Lawrence, Kan, 2011; 436–439.
66. Galeotti JP, Fraser G, Arcuri A. Improving search-based tset suite generation with dynamic symbolic execution. *Proceedings of the International Symposium on Software Reliability Engineering,* Pasadena, CA, 2013; 360–369.
67. Burnim J, Sen K. Heuristics for scalable dynamic test generation. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering,* L'AQUILA, ITALY, 2008; 443–446.
68. Xie T, Tillmann N, Halleux P, Schulte W. Fitness-guided path exploration in dynamic symbolic execution. *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks,* Estoril, Lisbon, Portugal, 2009; 359–368.
69. Aho AV, Lam M, Sethi R, Ullman JD. *Compilers, Principles, Techniques, and Tools*, 2nd edn. Addison-Wesley: Boston, MA, 2007.
70. Pande H, Landi W, Ryder BG. Interprocedural def-use associations in C programs. *IEEE Transactions on Software Engineering* 1994; **20**(5):385–403.
71. Sinha S, Harrold MJ, Rothermel G. Interprocedural control dependence. *ACM Transactions on Software Engineering and Methodology* 2001; **10**(2):209–254.
72. McMinn P. Search-based software test data generation: a survey. *Journal of Software Testing, Verification, and Reliability* 2004; **14**(2):105–156.
73. Tonella P. Evolutionary testing of classes. *Proceedings of the International Symposium on Software Testing and Analysis,* Boston, Massachusetts, 2004; 119–128.
74. Wappler S, Lammermann F. Using evolutionary algorithms for the unit testing of object-oriented software. *Proceedings of the Genetic and Evolutionary Computation Conference,* Washington, D. C. USA, 2005; 1053–1060.
75. Do H, Elbaum SG, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering: An International Journal* 2005; **10**(4):405–435.
76. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments on the effectiveness of data flow- and control flow-based test adequacy criteria. *Proceedings of the International Conference on Software Engineering,* Sorrento, Italy, 1994; 191–200.
77. Xie Q, Memon AM. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology* 2007; **16**(1):1–36.

DOI: 10.1002/stvr

78. Cai X, Lyu MR. The effect of code coverage on fault detection under different testing profiles. *Proceedings of International Workshop on Advances in Model-Based Testing,* St. Louis, Missouri, 2005; 1–7.
79. Frankl PG, Iakounenko O. Further empirical studies of test effectiveness. *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering,* Lake Buena Vista, FL, USA, 1998; 153–162.
80. Namin AS, Andrews JH. The influence of size and coverage on test suite effectiveness. *Proceedings of the International Symposium on Software Testing and Analysis,* Chicago, IL, USA, 2009; 57–68.
81. Piwowarski P, Ohba M, Caruso J. Coverage measurement experience during function test. *Proceedings of the 15th International Conference on Software Engineering,* Baltimore, Maryland, USA, 1993; 287–301.
82. Do H, Rothermel G, Kinneer A. Empirical studies of test case prioritization in a JUnit testing environment. *Proceedings of the International Symposium on Software Reliability Engineering,* Saint-Malo, Bretagne, France, 2004; 113–124.
83. Rothermel G, Untch R, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* October 2001; **27**(10):929–948.
84. Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 2001; **43**(14):841 –854.
85. Arcuri A. It does matter how you normalise the branch distance in search based software testing. *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation,* Paris, France, 2010; 205–214.
86. Arcuri A, Iqbal Z, Briand L. Formal analysis of the effectiveness and predictability of random testing. *Proceedings of the International Symposium on Software Testing and Analysis,* Trento, Italy, 2010; 219–229.
87. Dowdy S, Wearden S, Chilko D. *Statistics for Research*, 3rd edition. Wiley: Hoboken, NJ, USA, 2004.
88. Sthamer H. The automatic generation of software test data using genetic algorithms. *Ph.D. Thesis*, 1996.
89. McMinn P, Binkley D, Harman M. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology* 2009; **18**:11:1–11:27.

DOI: 10.1002/stvr