Hybrid Directed Test Suite Augmentation: An Interleaving Framework

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

*Department of Computer Science, Korea Advanced Institute of Science and Technology

Email: kimyunho@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

†Department of Computer Science and Engineering, University of Nebraska - Lincoln

Email: {zxu,myra,grother}@cse.unl.edu

***Abstract*—Test suite augmentation techniques generate test cases to cover code missed by existing regression test suites. Various augmentation techniques have been proposed, utilizing several test case generation algorithms. Research has shown that different algorithms have different strengths, and that combining them into a single hybrid approach may be cost-effective. In this paper we present a framework for hybrid test suite augmentation that allows test case generation algorithms to be interleaved dynamically and that can easily incorporate new algorithms, interleaving strategies, and choices of other parameters that influence algorithm performance. We empirically study an implementation of this framework in which we use two test case generation algorithms and several algorithm interleavings. Our results show that specific instantiations of our framework can produce augmentation techniques that are more cost-effective than others, and illustrate tradeoffs between instantiations.**

I. Introduction

When software engineers use regression testing to validate evolving systems, they often begin by running existing test cases. This may not be adequate, however, because code modifications can add new functionality and alter test coverage. *Test suite augmentation techniques* (e.g., [1], [29], [33], [39]) address this problem, by identifying where new test cases are needed (i.e, code elements in the new program that are new or affected by changes – hereafter referred to as "targets") and then generating them.

In prior work, we investigated several augmentation approaches, focusing primarily on test case generation techniques that reuse existing test cases. The first two approaches utilized concolic [39] and genetic algorithms [36], respectively. Empirical studies showed that each approach can be effective at covering code affected by changes.

In subsequent work [37], we compared augmentation techniques while varying factors that affect them, including the algorithm used to generate test cases, the order in which targets are considered while generating test cases, and the manner in which test cases are reused. Our studies showed that the primary factor affecting augmentation is the test case generation algorithm utilized; moreover, differences in algorithms lead them to possess different strengths and weaknesses. We reasoned that hybrid test case augmentation techniques could effectively combine these algorithms to produce approaches that are more cost-effective than any algorithms used singly.

In [38] we investigated this possibility further. We presented a simple *hybrid directed test suite augmentation technique* that operates by applying concolic test case generation to a program being regression tested, and then applies genetic test case generation. (We refer to this approach here as the *fixed interleaving* approach.) We presented empirical results showing that this approach can be more effective than non-hybrid techniques; however, in terms of efficiency the approach was disappointing, failing to outperform non-hybrid techniques.

In addition to problems with performance, another drawback of our fixed interleaving augmentation approach is that it assumes that test engineers will employ augmentation techniques to completion – that is, they will continue to try to augment coverage until no additional avenues for covering targets avail themselves. When time allows such testing, this assumption is reasonable. Frequently, however, regression testing is performed under time constraints. For example, in modern development practices, regression testing can be performed during maintenance phases of varying time limits, such as overnight or over the weekend. In such cases, the goal is not to find an augmentation technique which, at the end of its execution cycle, produces the best coverage of affected elements, but rather, to find an augmentation technique that *can achieve coverage more quickly* than others. No prior work has considered augmentation approaches from this standpoint.

In this paper we address both of the foregoing issues. We propose a framework for hybrid test suite augmentation that interleaves test case generation algorithms dynamically, that can easily incorporate new test case generation algorithms and interleaving strategies, and that can be parameterized to adjust target ordering and test case reuse approaches. While this framework is flexible enough to support the fixed interleaving approach, it also supports approaches that use finer-grained, *dynamic interleaving*. Under this framework, a controller *dynamically* makes decisions about which test case generation algorithm to choose at a given point in time, and based on this (with knowledge about each algorithm), it selects a set of targets to cover next and passes that with other necessary information to the selected algorithm. If the algorithm succeeds in attaining new coverage, it returns the new test cases and new coverage information back to the controller. In this way, the strengths of each test case generation algorithm can be leveraged dynamically over time.

We empirically study an implementation of this framework in which we apply the dynamic interleaving approach

to several versions of two non-trivial Unix utilities. In our study, we use two test case generation algorithms and several different algorithm interleavings. The results of our study show that our dynamic interleaving framework can be used to combine different test case generation algorithms, and that the dynamic interleaving approach that we evaluated almost always achieves higher branch coverage more quickly than the fixed interleaving approach.

II. Background and Related Work

A. Test Suite Augmentation

Let P be a program, let P' be a modified version of P, and let T be a test suite for P. Regression testing is concerned with validating P'. To facilitate this, engineers often begin by reusing T, and a wide variety of approaches have been developed for rendering such reuse more cost-effective via regression test selection (e.g., [28], [32]) and test case prioritization (e.g., [12], [23]). *Test suite augmentation* techniques, in contrast, are not concerned directly with reuse of T. Rather, they are concerned with the tasks of (1) *identifying affected elements* (portions of P' or its specification for which new test cases are needed), and then (2) *creating or guiding the creation of test cases that exercise these elements*.

Various algorithms have been proposed for identifying affected elements in software systems following changes (e.g., using models or specifications [4], program slicing or dependence graphs [3], [16], [31]). In this work we are concerned with generating test cases for these elements.

Appiwattanapong et al. [1] and Santelices et al. [33] combine dependence analysis and symbolic execution to identify chains of data and control dependencies related to changes, but present no specific algorithms for generating test cases. Person et al. (2008) [29] present an approach for program differencing using symbolic execution that can identify the effects of program changes and generate relevant test cases. Person et al. (2011) [30] use program analysis techniques to identify the parts of new programs that are affected by changes and apply symbolic execution to those. None of these approaches are integrated with reuse of existing test cases.

Only a few papers have considered augmentation from the standpoint of reusing and generating new test cases. We have already described our own work in this area [36]–[39], in which adaptations of genetic and concolic test case generation techniques use test resources and data from prior testing sessions to generate test cases to cover target code elements. More recently, Xie et al. [35] presented an approach for using dynamic symbolic execution to reveal execution paths that need to be retested, in which existing test cases can be utilized.

B. Automated Test Case Generation

There has been a great deal of research on techniques for automated test case generation. This includes work on generating test cases from specifications (e.g., [7]) from formal models (e.g., [19]) and by random selection of inputs (e.g., [8]). Several other techniques (e.g., [9]) use symbolic execution to find the constraints, in terms of input variables, that must be satisfied to execute a target path, and attempt to solve this system of constraints to obtain a test case for that path.

More recently, test case generation techniques that rely on dynamic information have appeared. Several such techniques use search-based approaches (e.g. evolutionary algorithms, tabu search, and simulated annealing) to generate test cases [2], [14], [26]. Other work (e.g., [6], [15], [34]) combines concrete and symbolic test execution to generate test inputs. This second approach is known as *concolic testing* or *dynamic symbolic execution*, and has proven useful for generating test cases for C and Java programs. In our work, we focus on these two classes of approaches, because they can make use of existing test cases, and because such test cases are readily available in a regression testing scenario. Here, we summarize these overall approaches.

1) Genetic Algorithms for Test Case Generation: Genetic algorithms (GAs) for structural test case generation have become common [14], [26]. GAs begin with an initial (often randomly generated) test case population and evolve the population toward targets that can be blocks, branches or paths in a program. Test inputs are represented in the form of a chromosome, and a fitness function is 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 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. The process is repeated until a stopping criterion has been met.

In the work of Fraser and Arcuri [14], the full test suite is targeted at once (the fitness function computes the quality of a test case relative to all branches rather just an individual branch). This helps the GA avoid spending too much time on infeasible or difficult-to-cover branches. While we target individual branches in this work, we include *serendipitous coverage* of additional branches [27]. As test cases are generated for the intended branch, some of the test cases cover other uncovered branches and we keep these test cases as well. We also use existing test cases as the starting point for our initial population, rather than random test cases. This reflects Fraser and Arcuri's observation that seeding strategies are important in test case generation using GAs [13].

2) Concolic Test Case Generation: Concolic testing (CT) [6], [15], [34] concretely executes a program while carrying along a symbolic state and performing symbolic execution of the path being executed. A symbolic path constraint gathered along the way is used to generate new inputs that drive the program along a different path on a subsequent iteration, by negating a predicate in the path constraint.

In the traditional application of concolic testing, test case reuse is not considered, and the focus of test case generation is on path coverage. First, a random input is applied to the program and the algorithm collects the path condition for this execution. Next, the algorithm negates the last predicate in

this path condition and obtains a new path condition. Calling a constraint solver on this path condition yields a new input, and a new iteration then commences, in which the algorithm again attempts to negate the last predicate. If the algorithm discovers that a path condition has been encountered before, it ignores it and negates the second-to-last predicate. This process continues until no more new path conditions can be generated. Ideally, the end result of the process is a set of test cases that cover all paths. (In practice, bounds on path length or algorithm run-time can be applied). In our application of the concolic approach (see [37]), we alter the foregoing process to work with existing test cases and we operate on an ordered list of targets, at the level of branch coverage.

3) Combinations of techniques: Recently, other researchers have combined test case generation techniques. Hybrid concolic testing [24] combines random and concolic test case generation. Inkumsah et al. [20] combine a genetic algorithm and concolic testing to generate test cases for programs. Borges et al. [5] use a meta-heuristic search technique to help symbolic execution solve complex mathematical constraints. Symbolic search-based testing [2] combines symbolic information with dynamic analysis to construct fitness functions that improve the efficiency of search-based testing for branch adequate test data generation. Malburg et al. [25] include constraint solving in the mutation stage of a genetic algorithm to ensure that mutated offspring efficiently explore different control flow in order to improve branch coverage. AUSTIN [22] uses heuristic search to cover primitive data type targets in C programs, and symbolic techniques for those with dynamic data structures. None of this work, however, addresses the test case augmentation problem.

Other researchers have also combined search-based and symbolic techniques for other purposes (for example, mutation testing [17]). The differences between that work and ours are that we focus on augmentation, and we do not modify the algorithms (or fitness), but instead we leverage the best algorithms by *dynamically interleaving* them at the right time. Our prior hybrid technique [38] also combines a genetic and a concolic algorithm, but this is done statically, in a fixed order (we do not use feedback during execution to select algorithms or target order). We compare this work with that in this paper.

III. Interleaving Framework

The results of our study of a fixed interleaving hybrid augmentation approach [38] show that when test case generation algorithms are interleaved, we can harness their different strengths to cover targets more effectively. The efficiency of the approach, however, was less (i.e., yielding longer run times) than that achieved by applying genetic or concolic algorithms individually. This suggests a need for improvements, particularly if our aim is to increase coverage more quickly in time-constrained environments. In addition, we want to allow more *dynamic decision-making* regarding how test case generation algorithms are interleaved, how targets are ordered, and how test cases are reused, and ultimately, we would like to seamlessly incorporate other test case generation algorithms.

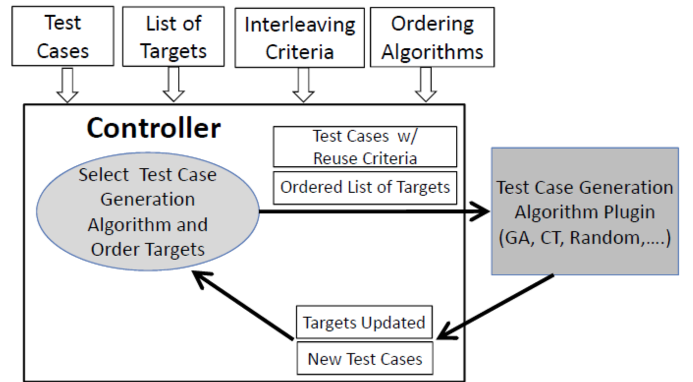

Fig. 1. Dynamic Interleaving Augmentation Framework

A. Dynamic Interleaving Augmentation Framework

Figure 1 depicts the architecture of our dynamic interleaving augmentation framework. A *Controller* is responsible for communication between test case generation algorithms and for selecting target orderings, test cases to reuse, and algorithm interleaving criteria. The input to the controller is the initial set of test cases, the set of targets that need to be covered, a set of criteria that tell the controller when to switch between test case generation algorithms, and test case selection and ordering techniques that are used to select next targets.

When the augmentation process begins we bootstrap the controller by having it call a specific test case generation algorithm. The controller selects targets for the algorithm and places them in an order. Then, it sends the ordered list of targets and test cases to the test case generation algorithm. After the test case generation algorithm finishes attempting to cover the given targets, it sends the results, along with updated coverage information and newly generated test cases, back to the controller. The controller then updates the coverage information and determines which algorithm to call next with which parameters, based on the current interleaving criterion.

There are many ways in which test case generation algorithms can be interleaved in our framework. Possible interleaving criteria include the execution time devoted to a particular algorithm, the number of targets covered, the number of targets remaining to be covered, or the number of serendipitous targets (targets not specifically targeted for coverage in a given invocation, but that happen to be covered anyway by generated inputs) covered.

B. Interleaving Criteria

In this work, we focus on interleaving criteria that consider numbers of uncovered branches. We say that an interleaving limit IL for test case generation algorithm A is calculated as the ratio of the number of previously uncovered targets that A has covered or attempted to cover in its current invocation to the number of previously uncovered targets that existed when A was invoked. For example, if A is invoked with a set of 100 uncovered targets, and IL is set to .25, then after A has covered or attempted to cover 25 of these uncovered targets, it has reached its interleaving limit. When an interleaving limit

IL has been reached, a test case generation algorithm ceases execution and returns control to the Controller, which then determines how to proceed next. The intuition behind this approach is that as IL decreases, the degree to which test case generation techniques are interleaved increases, and test case generation techniques take more frequent "turns" at generating test cases, which we believe will result in a reduction in overall test case generation time and increased coverage.

The maximum possible value for IL is 1.0. In [38], our fixed interleaving approach invoked the concolic test case generation algorithm first with IL set to 1.0, and then invoked the genetic test case generation algorithm with IL set to 1.0 (which caused the genetic algorithm to consider all remaining uncovered targets). In theory, values of IL can be as small as $1/|S|$, where $|S|$ is the total number of uncovered targets available for consideration when a test case generation algorithm is invoked; this value of IL corresponds to a single target. For simplicity, we denote this particular value of IL as $IL = 0$.

In this work, we vary IL to determine whether and how it affects the performance of dynamic interleaving test case augmentation. We consider the cases in which $IL = 0$ and $IL = 1.0$, and several values of IL in between these.

C. Test Case Generation Algorithms

We briefly describe the concolic and genetic test suite algorithms that we use in this work. See [38] for a more complete description and formal algorithms.

1) Concolic Test Suite Augmentation: In our algorithm, a *path condition* pc for a target program is a conjunction $b_{i_1} \wedge b_{i_2} \wedge ... b_{i_n}$ where $b_{i_1}, ... b_{i_n}$ are branch conditions in the target program and executed in order. Let $\bar{b}$ denote a paired branch of a branch b (i.e., if b is a `then` branch, $\bar{b}$ is the `else` branch). Let BR be the initial set of target branches. The algorithm repeats for each target branch $b_t \in BR$ that has not yet been covered. Initially, a set of new test cases, NTC, is empty and a set of target branches to cover, NBR, is set to BR. The start of the main procedure selects test cases that can reach $\overline{b_t}$ from within the set of test cases, TC. If there are no such test cases, the algorithm terminates. If there are such test cases, the algorithm obtains path conditions by executing the target program with the selected test cases From each obtained path condition pc the algorithm generates n_{iter} new path conditions as follows. Suppose the last occurrence of $\overline{b_t}$ is located in the mth branch of pc. Then, the algorithm generates n_{iter} new path conditions by negating $b_{i_m}, b_{i_{m-1}}, ..., b_{i_{m-n_{iter}+1}}$ and removing all following branches in pc, respectively.[1] If a newly generated path condition pc' has a solution tc_{new} (a new test case) and tc_{new} covers uncovered branches in NBR. NBR is then updated to reflect the new status of coverage and tc_{new} is added to the set of newly generated test cases NTC

2) Genetic Test Suite Augmentation: Our genetic algorithm is invoked for each target branch $b_t \in BR$ that has not yet been covered. It iterates for a number of generations or until b_t is covered. The first step calculates the fitness of all test cases in the test case population. Since the fitness of a test case depends on its relationship to the branch we are trying to cover, calculating the fitness requires that we run the test case. (For test cases provided initially we can use coverage information obtained while performing the prior execution of the test cases, TC, which in our case occurred in conjunction with determining affected elements.) Next, a selection is performed, which orders and chooses the best half of the chromosomes to use in the following step. The population is divided into two halves (retaining the ranking) and the first chromosome in the first half is mated with the first chromosome in the second half and this continues until all have been mated. Finally, a small percentage of the population is mutated, after which all test cases in the current population are executed.

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

D. Test Case Reuse Algorithms

We allow the concolic algorithm to reuse all newly generated test cases created during the prior executions of any test case generation algorithms. This is because our prior work [37] showed that using initial and newly generated test cases allows the concolic algorithm to achieve higher branch coverage than if it uses only initial test cases. We allow the genetic algorithm to reuse newly generated test cases within a limit relative to the given test case pool size. Our early work showed that, for the genetic algorithm, using newly generated test cases did not increase branch coverage much, but did increase execution time significantly.

E. Target Ordering Algorithms

In prior work [37], we found that a depth first ordering (DFO) of targets allowed the genetic test case generation algorithm to generate test cases more efficiently than a random ordering of targets. This ordering did not provide any benefits, however, to the concolic test case generation algorithm; for that algorithm, a random ordering of targets was equally effective as, but less expensive than, DFO. Therefore, in the instantiation of our approach that we study in this paper, we restrict our attention to DFO for the genetic algorithm and random ordering for the concolic algorithm.

IV. Empirical Study

Our goal is to empirically evaluate our framework, focusing on the relative cost-effectiveness of technique interleaving approaches, overall and in the presence of time constraints. We thus pose the following research questions.

RQ1: How do dynamic interleaving test suite augmentation techniques compare to one another and to the fixed interleaving approach in terms of effectiveness and efficiency.

RQ2: How does the performance of the test suite augmentation techniques considered differ, in terms of effectiveness, as time constraints differ?

TABLE I
STUDY OBJECTS

	# of total branches	# of uncovered branches
grep1	3934	1894
grep2	4146	2008
grep3	4234	2076
grep4	4262	2091
grep5	4264	2093
sed1	2646	1612
sed3	2918	1952
sed4	2918	1952

A. *Objects of Analysis*

To address our research questions we selected two non-trivial Unix utilities, `grep` and `sed`, from the SIR repository [11]. `grep` and `sed` are written in C and have multiple versions. The SIR repository provides test suites containing 790 and 359 test cases for the earliest versions of the programs, `grep0` and `sed0`, respectively. Test cases for `grep0` and `sed0` are applicable to all subsequent versions of the two programs. To investigate our approach in a regression testing setting, we utilized five versions of `grep`, `grep1` through `grep5`, and three versions of `sed`, `sed1`, `sed3`, and `sed4`.[2] Table I provides information about these study objects. The second column lists the total numbers of branches present in extended versions (as explained in Section IV-C1) of the programs, and the third column indicates the number of branches left uncovered in each version by the test suites. As the `grep` and `sed` versions increase, the number of uncovered branches typically increases, due to changes made to the code. Note that, although we do not explicitly analyze changes in new program versions in order to target them in testing, by considering the coverage of code in new versions by existing test cases, we are implicitly considering the impact of code modifications on testing.

B. *Variables and Measures*

Independent Variables. Our experiment manipulates two independent variables: augmentation technique and time constraint level.

IV1: Augmentation technique. We compare five augmentation techniques:

- A *fully dynamic interleaving* technique in which $IL = 0$. As noted earlier, this technique is maximally dynamic, in that it considers each test case generation technique relative to a single target before allowing the Controller to switch to another test case generation technique.
- Three intermediate levels of *dynamic interleaving techniques*, with IL set to .25, .50, and .75, respectively.
- A *fixed interleaving technique*, in which $IL = 1.0$.

To standardize the use of our framework in this study, we allow an algorithm to attempt to cover a target just once; therefore, once the concolic test case generation algorithm has attempted to cover all targets, it will not be selected again. By this approach, a given target branch can be considered at most twice – once by each algorithm.

IV2: Time constraint level. To consider time constraints we selected a basic time interval δ and partitioned the overall time in which techniques execute into sub-intervals of size δ. As a choice for δ we selected 5000 seconds, which partitions the total time required to execute the longest running of our techniques into 15 intervals.

Fixed Factors. To focus on the effects of interleaving, we chose fixed values for several additional factors that are parameterizable within our framework, as follows:

- As target orderings we use a random order for concolic test case generation, and DFO for genetic test case generation, as these are the orders that have been seen to be most cost-effective in prior studies [37].
- We allow the concolic algorithm to reuse any test cases that have been created in prior invocations of test case generation algorithms. We allow the genetic algorithm to reuse newly generated test cases within a fixed pool size.
- We use the concolic and genetic test case generation algorithms utilized in prior work [36], [39], and we always invoke the concolic test case generation algorithm first, as it has been shown to be the most efficient.

Dependent Variables. We wish to measure the efficiency and effectiveness of augmentation techniques when allowed to run to completion, and the effectiveness of techniques when restricted to limits imposed by time constraints. To do this we selected two variables and measures:

DV1: Efficiency in terms of time. To track augmentation technique efficiency when no time constraints are imposed, for each application of an augmentation technique we measured the wall clock time required to apply the technique up until the point at which it completes its execution.

DV2: Effectiveness in terms of coverage. The test case augmentation techniques we consider are intended to work with existing test suites to achieve higher levels of coverage in P' within the amount of time governed by a given time constraint. To measure the effectiveness of techniques within a time constraint C, we tracked the number of branches in P' that can be covered by each augmented test suite from the beginning of execution until time C. The effectiveness of a technique when run to completion is the coverage achieved when the technique completes its execution.

While our effectiveness variable concerns only coverage, when we assess coverage at time constraint intervals we are essentially assessing the *cost-effectiveness* of techniques relative to the costs represented by the given time intervals.

C. *Experiment Setup and Operation*

To establish the experiment setup needed to conduct our experiment we followed several steps.

1) Extended Programs: Our concolic test suite augmentation technique is implemented based on CREST [10]. CREST

[2] We excluded `sed2` and `sed5` because the prototype tool [18] that our genetic algorithm implementation relies on to generate control flow graphs cannot process them.

transforms a program's source code into an equivalent "extended" version in which each conditional statement with a compound Boolean condition is transformed into multiple conditional statements with atomic conditions without Boolean connectives (i.e., `if(`b_1 `&&` b_2`) f()` is transformed into `if(`b_1`) {if(`b_2`) f()}`). To integrate the concolic and genetic techniques into our interleaving framework, we opted to create extended versions of all our object programs, and apply all techniques to those versions.

2) Iteration Limits: Genetic algorithms iteratively generate test cases, and an iteration limit governs their termination. Similarly, the concolic algorithm that we use employs an iteration limit that governs the maximum number of path conditions that it should attempt to solve. After some exploratory trials, we found that 15 was a reasonable number of iterations for the genetic test suite augmentation technique and 11 was a reasonable number for the concolic test suite augmentation technique, because expanding the iteration limits beyond these numbers produces less than a 1% further increase in coverage in a substantial amount of time.

3) Tool Settings and Tuning: The Controller for our framework is implemented using 500 lines of BASH scripting code, and 1000 lines of Java code. For this study, we tuned our genetic algorithm by applying it directly to the extended object programs absent any existing suites. In our genetic algorithm, we limit the population size to 20 and we chose "best half" for selection. Our chromosomes and crossover are specific to the program input. For `grep` we divide the chromosome into three sections (one for the program options, one for the search pattern, and one for the file that is being searched). We apply one-point crossover to each of the first two sections independently (the input file is retained as-is). For `sed`, we consider each space-separated element of the input as a gene and perform a single one-point crossover on this chromosome.

We used the approach level in the fitness function, and 0.05 as a mutation rate. Mutation treats each gene and each character of the input as possible units of mutation and is performed evenly (with randomness) across the gene and input. For mutations performed on program options, we restrict the mutation to be from a valid set (we do not restrict mutations within the file characters). The genetic augmentation algorithm is implemented in Java and contains 4000 lines of code. For details on this implementation see [36]. For concolic testing, we used CREST-BV [21], which is an extended version of CREST that supports bit-vector symbolic path formulas. CREST-BV consists of 8400 lines of C++ code. For each object and technique, we executed each of our augmentation techniques five times, to control for randomness.

D. Threats to Validity

The primary threat to *external validity* involves the representativeness of our object programs and test suites. We have examined only five versions of `grep` and three of `sed`. While these are non-trivial, real C programs with actual versions, the study of other objects and other test suites may exhibit different cost-benefit tradeoffs. A second threat to external validity pertains to our algorithms; we have utilized only one variant of a genetic test case generation algorithm, and one variant of a concolic testing algorithm, and we have applied both to extended versions of the object programs, where the genetic approach does not require this and might function differently on the original source code. Subsequent studies are needed to determine the extent to which our results generalize.

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

Where *construct validity* is concerned, there are other metrics that could be pertinent to the effects studied. In particular, our measurements of cost consider only technique run-time, and omit costs related to the time spent by engineers employing the approaches.

E. Results

1) RQ1: Effectiveness and Efficiency of Techniques: Figure 2 presents the results obtained for the five versions of `grep` and the three versions of `sed`. In the graphs, the x-axes correspond to time in seconds and the y-axes correspond to newly covered branches. The five lines in the graphs track coverage achieved over time by each of the five techniques (with each line representing the average coverage measured per technique across five runs). The dotted and solid lines represent the results associated with the fully dynamic interleaving technique and the fixed interleaving technique, respectively. The dashed lines composed of short dashes, medium-length dashes, and long dashes represent the results of the dynamic interleaving technique when associated with IL settings of .25, .50, and .75, respectively (hereafter referred to as $IL.25$, $IL.50$, and $IL.75$).

The fully dynamic interleaving technique ultimately achieved the highest coverage among the five techniques on all versions of `grep` and all versions of `sed`. The IL.25 technique achieved the second highest coverage on all object programs. The IL.50 technique achieved the third highest coverage on all programs other than `grep1` and `sed3`. The fixed interleaving technique achieved the lowest coverage on all five versions of `grep` and on `sed4`. The fixed interleaving technique covered five more branches than the IL.75 technique on `sed1` and 13 more branches than the IL.50 technique on `sed3`.

The fully dynamic interleaving technique was the most efficient technique on all five versions of `grep` and `sed3`. The fully dynamic interleaving technique required longer to execute than IL.25 on `sed1` and required the longest execution time among all techniques on `sed4`. The fixed interleaving technique was the slowest technique on all programs other than `grep1` and `sed4`. Among the IL.25, IL.50, and IL.75 interleaving techniques, none clearly outperformed the others on our object programs. For example, the IL.25 technique

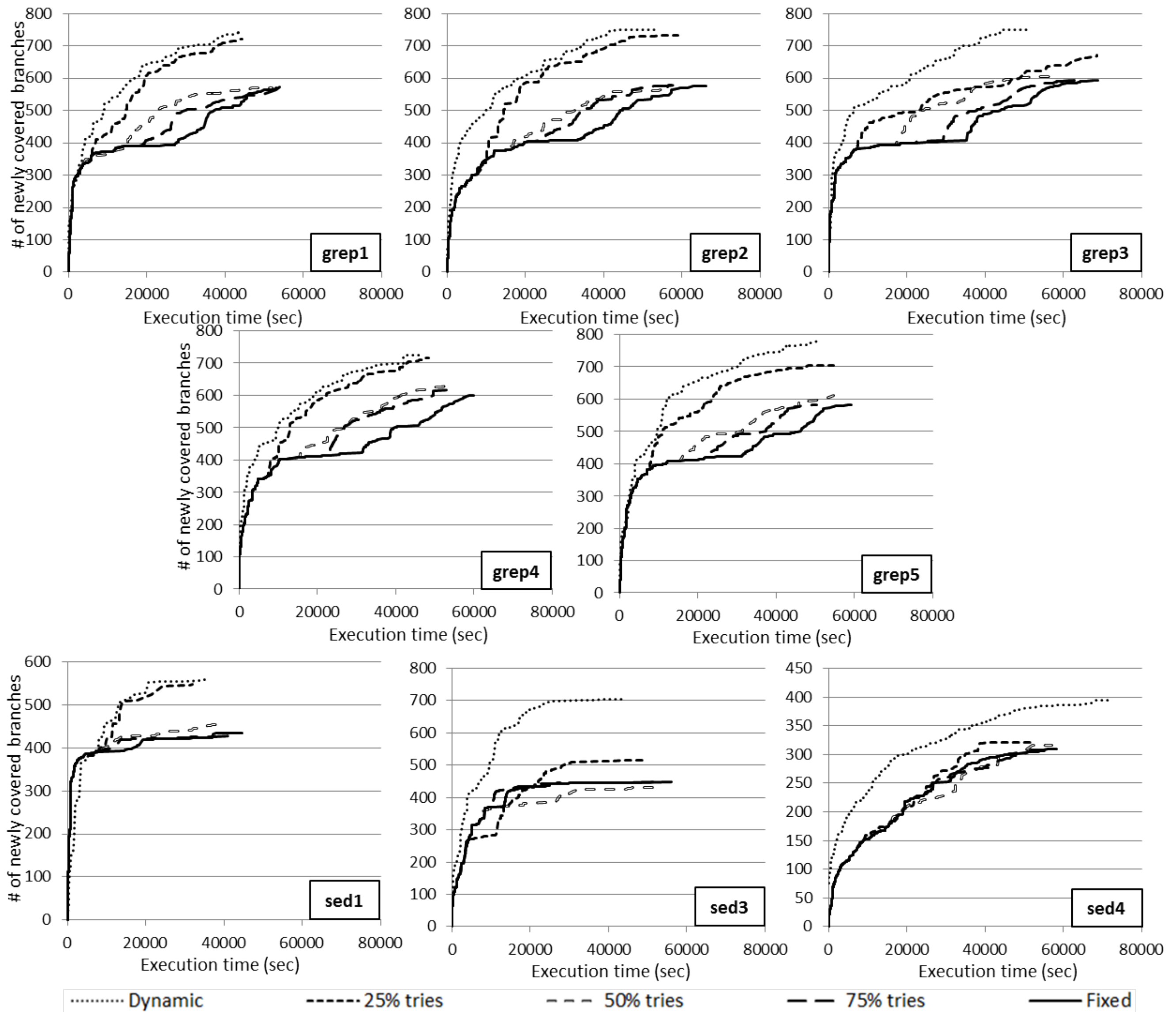

Fig. 2. Effectiveness and Efficiency of Hybrid Augmentation Techniques

required the longest time among the three interleaving techniques on `grep2` and `grep3`, but the IL.25 technique was the fastest of the three techniques on `grep1` and `grep4`.

As a further source of insight into the data, Table II shows the numbers of times in which the different augmentation approaches switched between concolic and genetic algorithms. The first column lists object program versions. Columns 2 – 6 represent the fully dynamic, IL.25, IL.50, IL.75, and fixed interleaving techniques, respectively. Considered alongside the graphs shown in Figure 2, increased switching between techniques does appear to be associated with increased final coverage achieved. This result likely occurs because the concolic and genetic algorithms are able to use test cases newly generated by prior invocations of the other algorithms.

TABLE II
CONTEXT SWITCHES BETWEEN CT AND GA

Programs	Dynamic	IL.25	IL.50	IL.75	Fixed
grep1	2466	51	22	9	1
grep2	2743	52	21	8	1
grep3	2841	51	23	9	1
grep4	2895	53	21	9	1
grep5	2787	54	21	7	1
sed1	2119	50	23	8	1
sed3	2489	53	20	8	1
sed4	2502	53	24	10	1
Average	2605.3	52.1	21.9	8.5	1.0

2) RQ2: Technique Performance Considering Time Constraints: We now consider results under different time constraints. The graphs in Figure 2 show that the fully dynamic interleaving technique achieved coverage fastest except during the earliest stages of execution of `sed1`. The IL.25 technique was the second most cost-effective, for almost all time periods on all programs, with exceptions occurring in the time period

TABLE III
EFFECTIVENESS OF AUGMENTATION TECHNIQUES OVER TIME CONSTRAINT LEVELS

Programs	Technique	Accumulated coverage across time constraint levels (5000 seconds per interval)														
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`grep1`	Dynamic	**414**	**524**	**579**	**642**	**667**	**696**	**703**	**734**	**745**	-	-	-	-	-	-
	IL.25	339	425	503	608	640	670	677	710	722	-	-	-	-	-	-
	IL.50	350	364	401	462	511	536	553	557	568	570	570	-	-	-	-
	IL.75	336	371	390	408	435	500	511	533	547	574	574	-	-	-	-
	Fixed	338	372	391	391	393	417	458	508	531	564	570	-	-	-	-
`grep2`	Dynamic	**436**	**500**	**570**	**609**	**656**	**680**	**705**	**734**	**750**	**751**	**752**	-	-	-	-
	IL.25	278	338	522	587	621	647	670	690	715	731	732	733	-	-	-
	IL.50	278	345	377	419	466	490	515	544	560	561	565	579	-	-	-
	IL.75	278	345	378	402	417	447	502	534	546	570	577	579	-	-	-
	Fixed	278	345	378	402	406	407	417	447	502	534	546	570	577	577	-
`grep3`	Dynamic	**482**	**526**	**560**	**586**	**636**	**661**	**702**	**728**	**750**	**752**	**752**	-	-	-	-
	IL.25	356	455	479	495	525	556	566	573	580	614	627	640	661	673	-
	IL.50	353	383	395	456	507	525	546	583	594	602	605	605	-	-	-
	IL.75	354	382	395	400	406	433	490	512	529	570	584	592	595	-	-
	Fixed	354	384	394	400	401	406	406	490	504	515	564	583	592	593	-
`grep4`	Dynamic	**438**	**499**	**553**	**615**	**640**	**675**	**692**	**699**	**724**	**724**	-	-	-	-	-
	IL.25	341	416	530	584	616	639	670	675	705	716	-	-	-	-	-
	IL.50	340	381	407	451	495	528	551	591	613	626	628	-	-	-	-
	IL.75	342	382	406	410	471	523	544	566	584	614	616	-	-	-	-
	Fixed	340	382	407	410	415	420	459	500	508	540	578	599	-	-	-
`grep5`	Dynamic	**416**	**512**	**617**	**653**	**681**	**698**	**735**	**744**	**765**	**773**	**781**	-	-	-	-
	IL.25	352	473	537	563	627	658	675	688	694	703	703	-	-	-	-
	IL.50	353	396	410	456	491	498	543	569	577	595	611	-	-	-	-
	IL.75	354	398	408	412	444	487	494	528	573	582	582	-	-	-	-
	Fixed	353	396	408	412	421	422	456	491	498	543	577	582	-	-	-
`sed1`	Dynamic	376	**457**	**509**	**527**	**554**	**555**	**558**	**558**	-	-	-	-	-	-	-
	IL.25	381	416	509	510	544	544	548	-	-	-	-	-	-	-	-
	IL.50	386	402	426	428	436	440	448	455	455	-	-	-	-	-	-
	IL.75	**388**	397	419	421	421	423	425	426	429	-	-	-	-	-	-
	Fixed	386	391	396	418	421	421	423	434	434	434	-	-	-	-	-
`sed3`	Dynamic	**416**	**512**	**617**	**672**	**697**	**699**	**701**	**705**	**705**	-	-	-	-	-	-
	IL.25	271	283	378	427	482	507	509	511	515	515	-	-	-	-	-
	IL.50	286	365	374	383	385	412	427	427	428	432	433	-	-	-	-
	IL.75	284	370	429	435	443	446	447	447	447	447	447	-	-	-	-
	Fixed	284	371	421	433	443	444	446	446	447	447	447	447	-	-	-
`sed4`	Dynamic	**195**	**237**	**277**	**301**	**315**	**326**	**345**	**356**	**368**	**380**	**385**	**387**	**387**	**394**	**394**
	IL.25	117	160	176	210	231	272	299	320	322	322	322	-	-	-	-
	IL.50	115	152	174	207	222	230	264	279	300	305	316	316	-	-	-
	IL.75	114	155	173	219	242	260	270	276	290	302	307	307	-	-	-
	Fixed	115	153	174	219	235	251	277	292	300	304	306	306	-	-	-

ranging from 37,000 to 48,000 seconds on `grep3`, and the time period ranging from 3,300 to 22,000 seconds on `sed3`. The IL.50 technique was the third most cost-effective on all versions of `grep` and `sed1`. The IL.50 technique was least cost-effective after 14,000 seconds on `sed3` and in the time periods from 22,000 to 43,000 seconds on `sed4`. The IL.75 technique followed behind the IL.50 technique. The least cost-effective technique was the fixed interleaving technique, in all time periods of all versions of `grep`, and `sed1`.

Table III presents the corresponding data more precisely. The first column lists the target programs and the second column lists the names of the augmentation techniques. Columns 3 – 17 represent the 15 time constraint levels, and the entries in each column list the cumulative number of branches covered up through the end of that time constraint level. Hyphens (-) in the table mean that the corresponding technique finished prior to that time period. The best coverage for each version and time period is displayed in bold font.

On all target programs except `sed1`, the fully dynamic interleaving technique achieved the highest coverage across all time constraint levels. On `sed1`, the fully dynamic interleaving technique also achieved the highest coverage across all time constraints levels except the first (i.e., the first 5,000 seconds), during which IL.50 covered 12 branches more.

V. DISCUSSION AND IMPLICATIONS

A. Performance of Dynamic Interleaving Techniques

As shown in Section IV-E, the fully dynamic interleaving technique achieved greater coverage more quickly than the IL.25, IL.50, IL.75, and fixed interleaving techniques. We believe that the primary reason for this difference in effectiveness is that the concolic test case generation algorithm used in the dynamic interleaving technique is able to better utilize new test cases generated by the genetic algorithm. As Xu et al. [37] report, utilization of new test cases can boost the effectiveness of the concolic approach in general. This cannot occur in the fixed interleaving technique because in this case, the concolic technique runs first and only once. For example, for `grep3`, concolic testing with the fixed interleaving technique covered 406.4 branches while concolic testing with the fully dynamic interleaving technique covered 562.2 branches (38.3% more, see Table IV). In the IL.25, IL.50 and IL.75 techniques, the concolic algorithm cannot utilize new test cases on the first 25%, 50%, and 75% of target branches considered, respectively, in its first run, and this decreases its effectiveness in these cases as well.

Regarding the improved *efficiency* of the dynamic interleaving technique we draw the following observations. The total

TABLE IV
TIME COSTS AND COVERED BRANCHES FOR `grep3`

Augmentation technique	Serendipitously covered branches	Tried targets	Time cost (secs)	Covered branches
Concolic in Dynamic	511.4	1433	24267.2	562.2
Genetic in Dynamic	151.6	1161	27048.8	190.2
Concolic in IL.25	411.4	1544	30620.6	471.2
Genetic in IL.25	148.8	1218	32315.4	194.8
Concolic in IL.50	365.6	1487	28224.0	422.0
Genetic in IL.50	145.2	1183	28712.0	182.6
Concolic in IL.75	361.2	1565	36137.4	410.6
Genetic in IL.75	151.0	1266	32570.6	188.0
Concolic in Fixed	355.4	1598	35063.2	406.4
Genetic in Fixed	147.8	1279	33873.8	188.2

TABLE V
EFFECT OF TEST POOL DIVERSITY ON THE GA

Program	Diversity		GA using Init. TCs		GA using Init. TCs + CT TCs	
	Init. TCs	Init. TCs + CT TCs	Time(s)	Cov.	Time(s)	Cov.
`grep1`	11.97	8.37	28394.4	221.4	30124.2	179.0
`grep2`	11.97	7.98	31745.6	214.4	35781.2	169.0
`grep3`	11.97	6.82	29878.8	219.8	33873.8	188.2
`grep4`	11.97	9.13	26984.2	227.6	28896.4	177.8
`grep5`	11.97	9.11	26374.4	193.0	28164.4	150.0
`sed1`	13.66	10.12	29856.4	58.8	31022.4	42.4
`sed3`	13.66	9.33	39672.6	86.2	41611.2	75.6
`sed4`	13.66	9.17	26113.0	67.0	27571.0	56.8

number of target branches that concolic test case generation attempted to cover in the fully dynamic interleaving technique was less than the number attempted by the IL.25, IL.50, IL.75 and fixed techniques, due to higher serendipitous coverage. Serendipitous coverage by the concolic algorithm in the fully dynamic interleaving technique was greater because the concolic algorithm utilizes the test cases generated in the prior execution of the genetic algorithm. Test cases generated by the genetic algorithm have different characteristics from test cases generated by the concolic algorithm, because the two algorithms use different approaches to generate test cases. For example, on `grep3`, serendipitous coverage achieved by the fixed interleaving technique was 1598, while for the dynamic interleaving technique it was 1433 (11.5% fewer).

Similarly, for the genetic algorithm, the total number of trials needed by the fully dynamic interleaving technique to cover targets was less than the number needed by the genetic algorithm in the fixed interleaving technique. For example, on `grep3`, the number of attempts to cover targets by the genetic algorithm in the fixed interleaving technique was 1279 while the number in the fully dynamic interleaving technique was 1161 (10.2% fewer, see Table IV). This difference occurs because concolic testing in the dynamic interleaving technique covers more branches than concolic testing in the fixed interleaving technique, leaving fewer branches to be covered by the genetic algorithm.

One exceptional case in the data regarding efficiency involves the IL.50 technique, whose execution time was less than that of the IL.25, IL.75, and fixed interleaving techniques. We believe that this occurred because the target branches on which the concolic and genetic algorithms required significant execution were covered serendipitously by the other algorithm.

From these observations, we can confirm that a *fully dynamic interleaving* of different test case generation techniques can improve both the effectiveness and efficiency of augmentation to a large degree.

B. Effect of Test Pool Diversity on the Genetic Algorithm

One possible reason for the low effectiveness of the fixed interleaving technique involves the diversity of the test case population. In the fixed interleaving technique we use initial test cases and test cases newly generated by the concolic algorithm to form an initial population of test cases for the genetic algorithm. The test cases generated by the concolic algorithm, however, tend to differ only slightly from existing test cases, because the concolic algorithm attempts to modify only the parts of a test case that are needed to generate a new test case executing the negated branch. Thus, it is more likely that an initial population of test cases for the genetic algorithm in the fixed interleaving technique will lack diversity, and this can reduce the effectiveness of the genetic algorithm. This argument can also be applied to the IL.25, IL.50 and IL.75 interleaving techniques, but to a lesser degree, because they begin with the concolic algorithm, and the generated test cases when the concolic algorithm tries a given $K\%$ of branches are passed to the next invocation of the genetic algorithm.

To investigate the effects of test case diversity on the genetic algorithm, we performed an additional set of experiments using the genetic approach with two different variants of test case reuse. First, we employed a genetic algorithm that uses only initial test cases on target branches that are not covered by the concolic algorithm. This setting is the same as that of the genetic algorithm in the fixed interleaving technique, but here, the genetic algorithm does not use test cases generated by the concolic algorithm. Second, we employed a genetic algorithm that uses initial test cases and test cases generated by the concolic algorithm. This second approach is the same as that of the genetic algorithm in the fixed interleaving technique. Note that in both approaches, the genetic algorithm does not try to cover branches already covered by the concolic algorithm.

Table V shows the diversity of test cases and results of the genetic algorithm. Column 1 indicates program and version names. Columns 2 and 3 show the diversity of the initial test cases and of the initial test cases plus test cases generated by the concolic algorithm, respectively. We calculated the sum of the pair-wise edit distances between test cases divided by the total number of possible test case pairs, and we use the resulting value as the diversity of test cases. Columns 4 and 5 show the results of the genetic algorithm using only initial test cases. Columns 6 and 7 show the results of the genetic algorithm using both initial test cases and test cases generated by the concolic algorithm.

As the data shows, test cases generated by the concolic algorithm decreased the diversity of the test pool (see the second and third columns), and decreased the effectiveness of the genetic algorithm (see the fifth and seventh columns). For example, in `grep1`, the use of test cases generated by the concolic algorithm decreased diversity from 11.97 to 8.37

(-30.1%, see the second and third columns in the first row) and decreased the number of newly covered branches from 221.4 to 179.0 (-19.2%, see the fifth and seventh columns in the first row).

From these results, we can confirm that, contrary to our initial expectations when configuring our techniques, the test cases generated by the concolic algorithm can decrease the effectiveness of the genetic algorithm by decreasing the diversity of the initial test pool for the genetic algorithm. In future work we intend to explore alternative reuse mechanisms for achieving greater diversity.

VI. Conclusions and Future Work

We have presented a dynamic interleaving framework for test suite augmentation, that can utilize different algorithms for test generation, while also being parameterizable for other factors potentially affecting the success of augmentation techniques. The core of our framework is a controller that selects the algorithm and next target at each iteration. We implemented a variant of this framework. Experiments on five versions of `grep` and three versions of `sed` show that a technique in which two test case generation algorithms are fully dynamic interleaved outperforms other techniques.

Future work includes adding algorithms to the framework, and experimenting with different orderings, different methods for providing or manipulating initial test suites, and additional software subjects. In particular, we intend to investigate whether the results of specific test case generation techniques have particularly high impact on others in particular circumstances, in order to better understand the synergies between them. We also plan to further investigate the time differential that was observed in our individual algorithms when covering different branches, and how that impacts the overall performance of the interleaving framework.

Acknowledgements

This work was supported by the AFOSR through award FA9550-10-1-0406 and by the NSF through awards CCF-1161767 and CCF-0747009 to the University of Nebraska - Lincoln. The work was also supported by the NRF Mid-career Research Program funded by the MSIP Korea (2012R1A2A2A01046172) and the MSIP (Ministry of Science, ICT & Future Planning) of Korea in the ICT R&D Program 2013.

References

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

[2] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *ASE*, 2011.

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

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

[5] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. Pasareanu. Symbolic execution with interval solving and meta-heuristic search. In *ICST*, 2012.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *CCCF*, Oct 2006.

[7] J. Chang and D. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *FSE*, Sept. 1999.

[8] T. Y. Chen and R. Merkel. Quasi-random testing. *TR*, 56(3), 2007.

[9] L. Clarke. A system to generate test data and symbolically execute programs. *TSE*, 2, May 1976.

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

[11] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESEJ*, 10(4), 2005.

[12] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *TSE*, 28(2), Feb. 2002.

[13] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *ICST*, 2012.

[14] G. Fraser and A. Arcuri. Whole test suite generation. *TSE*, 39(2), 2013.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, June 2005.

[16] R. Gupta, M. Harrold, and M. Soffa. Program slicing-based regression testing techniques. *JSTVR*, 6(2), June 1996.

[17] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *FSE*, 2011.

[18] M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith. Aristotle: A system for the development of program-analysis-based tools. In *ASC*, Mar. 1995.

[19] A. Hartman and K. Nagin. Model driven testing - agedis architecture interfaces and tools. In *CMDSE*, Dec. 2003.

[20] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE*, 2008.

[21] M. Kim, Y. Kim, and Y. Kim. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In *ICSE SEIP*, 2012.

[22] K. Lakhotia, M. Harman, and H. Gross. AUSTIN: A tool for search based software testing for the C language and its evaluation on deployed automotive systems. In *SSBSE*, pages 101–110, 2010.

[23] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *TSE*, 33(4), Apr. 2007.

[24] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[25] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *ASE*, 2011.

[26] P. McMinn. Search-based software test data generation: A survey. *JSTVR*, 14(2), 2004.

[27] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *TSE*, 27(12), Dec. 2001.

[28] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *FSE*, Nov. 2004.

[29] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE*, Nov. 2008.

[30] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, June 2011.

[31] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *ISSTA*, Aug. 1994.

[32] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2), Apr. 1997.

[33] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *ASE*, Sept. 2008.

[34] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, Sept. 2005.

[35] K. Taneja, T. Xie, N. Tillmann, J. Halleux, and W. Schulte. eXpress: Guided path exploration for regression test generation. In *ISSTA*, July 2011.

[36] Z. Xu, M. B. Cohen, and G. Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *GECCO*, July 2010.

[37] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *FSE*, Nov. 2010.

[38] Z. Xu, Y. Kim, K. M, and G. Rothermel. A hybrid directed test suite augmentation technique. In *ISSRE*, 2011.

[39] Z. Xu and G. Rothermel. Directed test suite augmentation. In *APSEC*, Dec. 2009.