University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

CSE Conference and Workshop Papers | Computer Science and Engineering, Department of

1-1-2009

Directed Test Suite Augmentation

Zhihong Xu
University of Nebraska - Lincoln, zxu@cse.unl.edu

Gregg Rothermel
Oregon State University, grother@cs.orst.edu

Follow this and additional works at: http://digitalcommons.unl.edu/cseconfwork

Part of the Computer Sciences Commons

Xu, Zhihong and Rothermel, Gregg, "Directed Test Suite Augmentation" (2009). *CSE Conference and Workshop Papers.* Paper 125.
http://digitalcommons.unl.edu/cseconfwork/125

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Conference and Workshop Papers by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

2009 16th Asia-Pacific Software Engineering Conference

Directed Test Suite Augmentation

Zhihong Xu and Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
{*zxu, grother*}*@cse.unl.edu*

***Abstract*—As software evolves, engineers use regression testing to evaluate its fitness for release. Such testing typically begins with existing test cases, and many techniques have been proposed for reusing these cost-effectively. After reusing test cases, however, it is also important to consider code or behavior that has not been exercised by existing test cases and generate new test cases to validate these. This process is known as *test suite augmentation*. In this paper we present a *directed test suite augmentation technique*, that utilizes results from reuse of existing test cases together with an incremental concolic testing algorithm to augment test suites so that they are coverage-adequate for a modified program. We present results of an empirical study examining the effectiveness of our approach.**

***Keywords*-regression testing, augmentation, concolic testing**

I. 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 existing test cases. In this context, retest-all methodologies [1], [2] re-use all of a system's previously developed test cases, while regression test selection techniques (e.g., [3]) attempt to reduce costs by re-using test cases selectively.

Reusing test cases is important, but having done so, engineers must also focus on code or system behavior which, due to changes, is not addressed by these. *Test suite augmentation techniques* (e.g., [4], [5]) help with this, by identifying where new tests are needed and creating them.

Despite the importance of testing new code and system behaviors, most research on regression testing has focused primarily on test suite reuse rather than test suite augmentation (see Section II). There has been some research on reuse that has also led to algorithms for *identifying where new test cases are needed*, but these approaches do not then generate test cases, leaving that task to engineers [4], [6], [7], [8], [5]. There has been some research on generating test cases given pre-supplied coverage goals [9], [10], [11], but the application of that to augmentation has not been explored, and the approaches have not been integrated with techniques for identifying where testing is needed.

In this paper we propose a *directed test suite augmentation technique*, that integrates an existing regression test selection algorithm [3] with an adaptation of a concolic approach [12] for test generation. Together, these techniques integrate approaches to test reuse and augmentation, leveraging test resources and data obtained from prior testing sessions for both tasks. We present results of an empirical study examining the performance of our approach. Our results show that the approach can be effective and efficient.

II. Background and Related Work

A. Regression Testing

Let P be a program, P' be a modified version of P, and T be a test suite for P. Regression testing is concerned with validating P'. To facilitate this, engineers may use the *retest-all* technique [1], re-executing all viable test cases in T on P', but this can be expensive. *Regression test selection* (RTS) techniques (e.g., [6], [3], for a survey see [13]) use information about P, P' and T to select a subset T' of T with which to test P'. *Safe* RTS techniques (e.g. [3]) guarantee that under certain conditions, test cases not selected could not have exposed faults in P' [13]. Empirical studies have shown that these techniques can be cost-effective.

In this work we use one particular safe RTS technique, `Dejavu` [3], to help drive test suite augmentation. `DejaVu` performs simultaneous depth-first traversals on control flow graphs (CFGs) for procedures in P and P' to find *dangerous edges* that lead to code that has changed. *Execution traces* of test cases (bit vectors indicating whether edges were taken) on the old version of P are then used to select test cases that traversed dangerous edges in P.

To illustrate, consider program `foo` and modified version `foo'`, whose control flow graphs are shown in Figure 1 with a node corresponding to modified code shaded in the CFG on the right. To select test cases from an existing test suite for `foo`, `DejaVu` constructs these CFGs and begins traversing them synchronously and depth-first along identically labeled edges, comparing the code associated with nodes in the two graphs for equivalence. As long as this traversal finds only identical code in the two versions, as occurs in the example along paths (E, P1, S4, X), no changes are encountered and no test cases need to be selected. Such identical paths through the code constitute *safe spaces* through which tests can pass unaffected by code changes. When non-identical code is found, however, as on path (E, P1, P2), all test cases known to have reached the dangerous edge leading to the

1530-1362/09 $26.00 © 2009 IEEE
DOI 10.1109/APSEC.2009.29

IEEE computer society

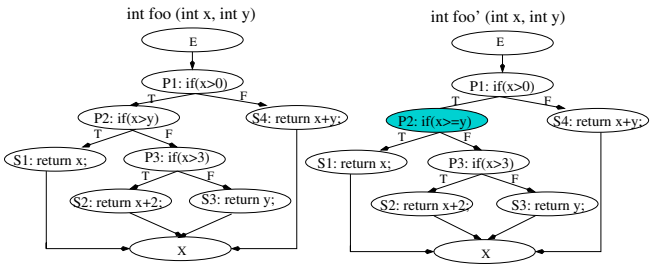

Figure 1. CFGs for two program versions

differing node (here, P1→P2) on the prior execution of T on P are selected.

B. Test Suite Augmentation

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

Several techniques have been proposed for identifying coverage requirements related to code changes [6], [7], [8]. These techniques use various dependence analyses, such as slicing on program dependence graphs, to select existing test cases that should be re-executed, while also identifying portions of the code that are related to changes and should be covered by tests. However, these approaches do not provide associated methods for generating actual test cases to cover the identified code.

Three recent papers [4], [14], [5] specifically address test suite augmentation. Two of these [4], [5] present an approach that combines dependence analysis and symbolic execution to identify test requirements that are likely to exercise the effects of changes, using specific chains of data and control dependencies to point out changes to be exercised. A potential advantage of this approach is a fine-grained identification of coverage needs; however, the papers present no specific algorithms for generating test cases. The third paper [14] presents a more general approach to program differencing using symbolic execution, that can be used to identify requirements more precisely than [4], [5], and yields constraints that can be input to a solver to generate test cases for those requirements. However, this approach is not integrated with reuse of existing test cases.

C. Concolic Testing

Concolic testing (concolic execution) [15], [16], [12] extends the semantics of a program by concretely executing the program, while carrying along a symbolic state and simultaneously performing symbolic execution of the path that is being executed. It then uses the symbolic path constraint gathered along the way to generate new inputs that will drive the program along a different path on a subsequent iteration, by negating a predicate in the path constraint. In this way, concrete execution guides the symbolic execution and replaces complex symbolic expressions with concrete values when needed to mitigate the incompleteness of the constraint solvers [12]. Conversely, symbolic execution helps to generate concrete inputs for the next execution to increase coverage in the concrete execution scope.

We explain the process briefly. First, concolic testing uses a random input to invoke 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, although in practice the approach is limited by the powers of constraint solvers.

The foregoing discussion concerns the application of concolic testing to single procedures, however, [12] extends the technique to the interprocedural level. Other subsequent concolic testing applications [17], [18] also apply this technique at the interprocedural level.

III. Directed Test Suite Augmentation

A. Algorithm

When program P evolves into P', coverage of P' by a prior test suite T can be affected in various ways. Some new code in P' may simply not be reached by test cases in T, and some test cases in T may take new paths in P', leaving code that was previously covered in P uncovered. RTS techniques can help select those test cases in T that encounter code changed for P', and thus may take different paths in P'. We use these techniques to indicate such test cases. We then use information gathered previously for test cases in T to generate test cases that cover uncovered code to form a branch coverage test suite T' for P', using a modified concolic testing approach. We now discuss our approach as applied intraprocedurally (to single procedures).

Algorithm 1 presents our algorithm, DTSA. The *main procedure* of DTSA (lines 1-6), consists of three steps. Step 1 uses the `Dejavu` RTS technique to partition test suite T into two subsets, one containing *affected* test cases (test cases that reach dangerous edges) and one containing *unaffected* test cases (test cases that do not reach dangerous edges). Step 2 reruns the affected test cases, and calculates a *testing objective* which includes all of the branches in P' that need to be covered. Finally, based on information retrieved from prior executions of unaffected test cases and executions

Algorithm 1 DTSA

Require: Set T of test cases for P
CFG_p, P's control flow graph
$CFG_{p'}$, P''s control flow graph
Ensure: Set T' of test cases for P'
Declare: Set $Goalset$ of branches in P' to be covered

1: **Main Procedure**
2: $Goalset$=**RTS**
3: $Goalset$=**RerunAffected**
4: **if** $Goalset \neq \emptyset$ **then**
5: call **Augment**
6: **end if**
7:
8: **Procedure RTS**
9: call `Dejavu` to find affected test cases and update unaffected test cases' trace information and path conditions in P'
10: subtract branches in $CFG_{p'}$ covered by unaffected tests to form $Goalset$
11:
12: **Procedure RerunAffected**
13: rerun all affected test cases and gather their trace information and path conditions
14: subtract branches in $CFG_{p'}$ covered by affected test cases from $Goalset$
15:
16: **Procedure Augment**
17: $Predicatehit$=PickPredicatehit($Goalset$,$CFG_{p'}$)
18: order branches in $Predicatehit$
19: **for** each $e_j \in Predicatehit$ **do**
20: find all test cases covering the source of e_j
21: use their path conditions to do $DelNeg$ at e_j's source
22: **if** path conditions after $DelNeg$ have not been seen before **then**
23: call $ConstraintSolver$ to solve them
24: **if** they are solvable **then**
25: put them into T'
26: run new generated test cases to obtain trace information, path conditions and coverage information
27: **if** they cover any branches in $Goalset$ **then**
28: subtract them from $Goalset$
29: update $Predicatehit$ according to $Goalset$
30: **end if**
31: **end if**
32: **end if**
33: **end for**

of affected test cases, Step 3 attempts to generate test cases to cover the branches in the testing objective.

Procedure *RTS* (lines 8-10) summarizes Step 1. The algorithm invokes `Dejavu` to find the sets of affected and unaffected test cases. We extend `Dejavu` to also find the corresponding unaffected test cases' trace information, path conditions and covered branches in P' as it synchronously traverses the CFGs, a process that succeeds because the traces and condition information that need updating all exist prior to code changes and can be found as `Dejavu` traverses the graphs. Next, the algorithm (line 10) subtracts the branches covered by the unaffected test cases from $CFG_{p'}$, placing remaining branches into *Goalset*.

Procedure *RerunAffected* (lines 12-14) summarizes Step 2. The procedure reruns all affected cases that are selected by `Dejavu` to allow engineers to verify their outputs; during this re-execution, trace and path condition information for these test cases are also collected. If an affected test case covers branches in *Goalset*, the branches it covers are subtracted from that set. After all affected test cases have been run, control returns to the *Main Procedure* which then checks whether $Goalset$ is empty (line 4). If it is, then our test suite is branch coverage adequate for P' and the algorithm terminates; otherwise, the algorithm continues.

The third and most significant step is procedure *Augment* (lines 16-33). Based on information gathered in the first two steps, the algorithm attempts to augment T using a concolic testing approach. The step begins (line 17) by locating, in $Goalset$, the branches for which the source node is a predicate node that is covered by at least one existing test case; these become the immediate targets for test generation. (These branches are ordered in line 18 for optimization reasons; we explain this later.)

The algorithm next enters a loop in which it selects branches one by one. For a given branch e_j with source (predicate) node p, the algorithm tries all path conditions for test cases whose execution traces reach p. For each such path condition, the algorithm deletes all predicates following p and negates p (the $DelNeg$ operation in line 21) to generate another path condition. If the generated path condition has not been seen before, the algorithm uses it to generate a new test case. Otherwise, the algorithm ignores it and moves on to the next path condition.

By calling a constraint solver to solve a modified path condition, the algorithm may obtain a test case to cover e_j. This test case and its trace and path condition information are saved. If the test case's trace covers branches in $Goalset$, $Goalset$ and $Predicatehit$ are updated to indicate the new coverage. If the solver cannot solve the path condition, the algorithm considers other path conditions that cover the predicate. If all path conditions fail the branch may be unreachable, or it is reachable and other methods will need to be found to generate test cases to reach it.

Two aspects of DTSA that differentiate it from existing instantiations of concolic testing bear further discussion. First, the algorithm iterates through all path conditions whose execution traces reach p (line 20) instead of stopping when a test case has been generated for the initial target branch e_j. It does this because doing so may allow it to generate more test cases to reach predicates following e_j, which may control additional branches needing to be covered. This increases the possibility of covering branches that are later in flow.

Similar reasoning motivates the branch ordering that occurs in line 18. Test cases execute CFG edges from predicates that are reached earlier to those that are reached later, and thus, passing through earlier branches is a precondition to reaching later branches; achieving coverage of earlier predicates leads automatically to coverage of certain later ones, and also produces test cases whose path conditions that can be manipulated to generate new test cases to cover later branches. Thus, we order the branches in $Predicatehit$ in breadth first order prior to using them.

B. Example

We use an example to illustrate how the algorithm works. Suppose we have five test cases for program `foo` in Figure

1, t_1=($x = 2$, $y = 2$), t_2=($x = 4$, $y = 4$), t_3=($x = 1$, $y = 0$), t_4=($x = 4$, $y = 3$), t_5=($x = -1$, $y = 0$), which are adequate for branch coverage in `foo` but not in `foo`$'$ due to the change in the second predicate.

In Procedure *RTS*, t_1, t_2, t_3 and t_4 are selected as affected test cases, since their traces contain the predicate node P2, whose content has changed. Test t_5 is treated as unaffected and it also covers branches (P1, S4). $Goalset$ contains (P1, P2), (P2, S1), (P2, P3), (P3, S2) and (P3, S3).

In Procedure *RerunAffected* for P', test cases t_1, t_2, t_3 and t_4 are rerun and their traces are obtained, all of which are (E, P1, P2, S1, X). After subtraction of the branches covered by these, $Goalset$ contains (P2, P3), (P3, S2) and (P3, S3). Since P2 is covered by existing test cases, $Predicatehit$ includes (P2, P3). Four test cases' executions exercise $P2$, so the algorithm enters line 21 to use their path conditions one by one to attempt to generate new test cases.

First, t_1's path condition, ($x > 0 \wedge x \geq y$), is selected. $DelNeg$ is applied to P2, obtaining another path condition, ($x > 0 \wedge x < y$). Using the solver to solve it, a new test case is produced, t_6=($x = 1$, $y = 2$), that covers branches (P2, P3) and (P3, S3). At the same time, one more path condition, ($x > 0 \wedge x < y \wedge x \leq 3$), is collected. Since this path covers some branches in $Goalset$, $Goalset$ and $Predicatehit$ are updated. Now $Goalset$ has one branch left, (P3, S2), and $Predicatehit$ contains one branch, (P3, S2), since P3 is covered by t_6. The algorithm also uses the path conditions for t_2, t_3 and t_4 to generate new test cases. Since these have the same path conditions as t_1 after $DelNeg$ is applied to P2, the algorithm ignores them. Using (P3, S2) from $Predicatehit$ as the next objective, the algorithm enters the next iteration. Running $DelNeg$ on predicate P3 of t_6's path condition, another path condition, ($x > 0 \wedge x < y \wedge x > 3$), is produced. By solving this, the algorithm obtains an input, $t7$=($x = 4$, $y = 5$), to cover branch (P3, S2). After updating $Goalset$, it becomes empty. At this point, the algorithm has generated test data covering all branches in `foo`$'$.

C. Extension to Interprocedural

Thus far we have presented our approach at the intraprocedural level, but as mentioned in Section II, concolic testing has also been extended to function interprocedurally. Following similar extensions we extended our technique to the interprocedural level as well. The algorithm remains essentially as presented above, however, in addition to ordering branches within methods (line 18) we use depth first ordering to order methods based on the program's call graph, ensuring that branches in callers are covered first.

D. Implementation

We implemented our algorithms within the `Sofya` analysis system [19], which provides utilities for code instrumentation and CFG construction. We used the RTS module, `Dejavu`, provided with `Sofya`, to find affected and unaffected test cases. With the help of the Soot framework [20], we inserted code into P and P''s source code to obtain the path condition for each execution. With CFGs and trace information, coverage information was obtained. Then we built a concolic testing module to use trace information to target uncovered branches and generate new test cases.

IV. Empirical Study

To provide initial data on the potential applicability of our DTSA approach we conducted an empirical study. The research questions that we address are:

- **RQ1**: How efficient is DTSA at generating test cases to complete the coverage of P'?
- **RQ2**: How effective is DTSA at generating test cases to complete the coverage of P'?

The remainder of this section describes our objects of analysis, variables and measures, experiment setup, results, and threats to validity.

A. Objects of Analysis

Since our implementation functions only on programs that utilize arithmetic operations, as objects for our experiment we use 42 versions of one of the Siemens program, `Tcas`, which is available from the SIR repository [21]. `Tcas` includes an original version and 41 revised (faulty) versions, which we denote here as v_0 and v_k ($1 \leq k \leq 41$), respectively. The program is also equipped with a "universe" of 1608 distinct test cases, consisting of black and white box tests, and representing a population of potential test cases. Because `Tcas` was originally written in C and our implementation of DTSA functions on Java programs, we converted all of the versions of `Tcas` to Java, as was done in [4]. The Java versions of `Tcas` have two classes, 10 methods and about 200 non-comment lines of code.

In practice when programs evolve, some test suites may need to be augmented while others may not. Therefore, in our study we utilize 1000 distinct test suites for v_0. While test suites are available in the SIR repository for the C version of `Tcas`, those suites were not coverage-adequate for the Java version. Thus, we employed the same greedy strategy utilized to produce the test suites for the C version to our Java version to create branch-coverage-adequate suites: randomly and greedily selecting test cases from the universe and adding them to the suite as long as they add coverage, and continuing until all reachable branches are covered.

B. Variables and Measures

1) Independent Variables: As independent variables we wish to consider our DTSA technique, and an alternative control technique. One such control technique could be found in existing augmentation techniques; however, as discussed in Section II, all such existing techniques merely identify coverage requirements, leaving the creation of test

cases to humans. Studies involving humans are expensive, and before conducting such studies it is reasonable to first determine whether our approach can be applied efficiently and effectively. As a control technique in this case, it makes sense to compare the approach to one in which, given P', concolic testing is reapplied from scratch with a goal of achieving branch coverage. Such a comparison allows us to assess the cost-benefit tradeoffs, in efficiency and effectiveness, that can be achieved by DTSA through its reuse of test cases.

Our independent variable thus consists of two techniques: the DTSA technique described in Section III and the basic concolic testing technique described in Section II, modified to operate on branch coverage.

In our implementation of concolic testing, when we run a test case we record its associated path condition, and then we apply the *DelNeg* operation for all input-related predicates, attempting to generate modified path conditions that will lead to coverage of as many branches as possible. We use Yiecs [22] to solve these modified path conditions, yielding new test cases that cover uncovered branches. For each new test case we repeat this process, until we have utilized all test cases. We record all of the path conditions that have been used and ignore duplicates. When we apply the *DelNeg* operation to a predicate, if both branches are already covered, we ignore the modified path condition too. Ultimately, for each new version, this process yields a test suite that covers all reachable branches possible.

2) Dependent Variables and Measures: We chose two dependent variables and corresponding measures to address our research questions. The first variable relates to costs of the techniques, and the second measures the effectiveness of the techniques in generating test suites. These measures help us understand the general performance of the two techniques, in a manner that provides guidance on their relative strengths and weaknesses.

Technique cost.: To measure technique cost, one approach would be to measure execution time. However, with prototype implementations and studies of comparatively small applications this measure is not an appropriate indicator of the costs in practice.

An alternative approach to cost measurement involves tracking the number of invocations, by techniques, of the operations that most directly determine technique cost. For the techniques that we consider the operation that matters most involves the solution of constraints. Thus, in this study, we measure the number of constraint solver calls made by the techniques.

Technique effectiveness.: We have chosen attainment of branch coverage as our test suite generation objective, and both of our techniques target it. For both techniques, however, there are limitations in achieving full branch coverage. When we use DTSA to generate test cases to cover all branches, we are limited by the existing test cases, and using these we may be unable to generate test cases that cover certain branches. In concolic testing, operations focus on predicates and on achieving coverage of these may omit generating additional test cases that could otherwise achieve coverage beneath these.

Given the foregoing, a measure of technique effectiveness involves its ability to generate coverage-adequate test suites, and thus, we track that coverage.

C. Experiment Setup

There are several issues regarding the setup for the experiment that need to be clarified. First, we conducted our experiments using v1.5.2 of the Java Runtime Environment (JRE) in a Linux environment. For consistency, all measurements for our object program were collected on the same system, a Pentium-M 1600 Mhz system with 1 Gb RAM running SuSE Linux 11.1.

Second, concolic testing can fare differently on different runs, depending on the inputs it randomly chooses initially. DTSA execution can fare differently on given test suites. Thus, it is important to compare data for both techniques on multiple executions, and on DTSA using multiple test suites. Accordingly, to conduct our study, for each version of the object program considered, and for each test suite augmented for that program, we also conducted a run of concolic testing on a randomly generated set of initial inputs. This procedure also ensured equal numbers of runs of the two techniques, facilitating subsequent analysis.

Third, all code-coverage-based testing techniques face issues involving infeasible code components – components (e.g., branches, statements, and so forth) that cannot be reached on any executions and thus cannot be covered. Adequacy criteria are not required to cover infeasible components, and coverage adequacy is measured in terms of percentages of feasible components covered. Each version of our object program has some unreachable branches; we determined these by inspection and we measure coverage in terms of the feasible branches only.

Finally, given the versions of our programs and the test suites created for them, there are numerous cases in which test suites applied to changed versions do *not* leave reachable branches uncovered. These are cases where augmentation is *not* needed. We omit these cases and gather data for just the instances in which augmentation is necessary.

Given the foregoing, to conduct our study we performed the following two steps. First, we instrumented and created the CFG for v_0. We then executed v_0 on each of our 1000 test suites, collecting test trace information and path conditions for each test case in each suite, for use in the next step. Second, for each new version v_k ($0 < k \leq 41$) of v_0, we constructed the CFG for v_k. Then, for each version, for test suite T_k ($0 < k \leq 1000$), we performed the following steps. (1) Executed all test cases in T_k on v_k. (2) Ran algorithm DTSA on the CFGs for v_0 and v_k,

together with the saved test trace and path condition data for v_0. If T_k needs to be augmented the algorithm performs the augmentation step, and we save the required data on performance. (3) If T_k needed to be augmented in the prior step we also perform a run of concolic testing on v_k, starting from randomly generated program inputs, saving the required data on performance.

D. Threats to Validity

The primary threat to external validity for this study involves the representativeness of our object program, versions, and associated test suites. We have examined only one software subject, coded in Java, and other systems may exhibit different cost-benefit tradeoffs. We have considered only one set of versions of this subject, all based on changes made to the initial version, and sequences of releases may exhibit different tradeoffs. Subsequent studies are needed to determine the extent to which our results generalize, and the extent to which the approach scales to larger systems.

The primary threat to internal validity for this experiment is possible faults in the implementation of the algorithms and in tools we use to perform evaluation. We controlled for this threat through extensive functional testing of our tools. A second threat involves inconsistent decisions and practices in the implementation of the two techniques studied; for example, variation in the efficiency of implementations of common functionality between techniques could bias data collected. We controlled for this threat by having these two techniques implemented, insofar as this was possible, by the same developer, utilizing consistent implementation decisions and shared code.

Where construct validity is concerned, there are other metrics that could be pertinent to the effects studied, such as the total execution cost of the two techniques. However, in this initial study our subject is not sufficiently complex, and our tools not sufficiently optimized for run-time, to render comparisons of execution times meaningful.

E. Results and Analysis

For our subject, we find 29 versions out of 41 versions needing to be augmented, because with the exception of unreachable branches they have other branches uncovered by old test suites. We analyze our data relative to those 29 versions for each of our research questions in turn.

1) Number of Constraint Solver Calls: To address RQ1 (efficiency of DTSA compared to efficiency of concolic) we compare the number of constraint solver calls made by the two techniques. Figure 2 presents boxplots that show the number of solver calls per technique for the 29 versions. The x axis enumerates each version and technique using a suffix of *D* to denote DTSA and a suffix of *C* to denote concolic testing.

As the boxplots show, in most cases the number of solver calls made by DTSA is substantially less than the number made by concolic testing. In some cases, however, as in v13 and v37, there is some overlap in the ranges of the data sets.

To formally assess which mean differences are statistically significant we used a paired t-test. Our hypothesis is that the number of constraint solver calls of DTSA will be less than that of concolic testing. We expect to find negative mean differences (that is, DTSA consistently has fewer calls to the solver than concolic testing on average) in our data. Mean differences in which the t-test ρ (rho) value is less than or equal to 0.05 are deemed statistically significant.

Table I presents the result of our analysis, providing the mean differences in numbers of solver calls between DTSA and concolic testing per version, and ρ-values from the t-tests. Versions for which results are not given are those in which only one test suite needs to be augmented, or, in the case of v15, where two pairs of the values are the same and a t-test cannot return a result. As the table shows, all of the mean differences are less than 0 and all computable ρ-values are less than 0.05, supporting our hypothesis.

2) Coverage Criteria: Next we explore RQ2, which involves the effectiveness of DTSA relative to concolic testing, in terms of achieving adequate branch coverage when augmenting test suites.

Table II displays the mean numbers of branches not covered by the test suites generated by the two techniques. The total number of reachable branches ranges from 79 to 84 for all versions and all of the branches listed in this table are reachable — infeasible branches were calculated by inspection and excluded. For most versions of our object program, concolic testing left about three branches uncovered, with exception of v9, v10, v11, v21 and v23. On the first three of these, five or six branches were left uncovered, while on the last two, over 10 were left uncovered. DTSA, in contrast, achieved 100% branch coverage on 17 versions, with an average of three on most other versions. On all but versions v10 and v14, however, DTSA achieved better coverage than concolic testing.

V. Discussion

Our results show that, for the object program and versions considered, the DTSA technique can be applied effectively, and more efficiently than a full application of concolic testing. In general, when using DTSA to do test suite augmentation, we are able to restrict our attention to a smaller number of testing objectives than full concolic testing, resulting in substantially fewer solver calls.

However, DTSA is not always more efficient than full concolic testing. In full concolic testing, it is possible to generate a single test case that covers several branches. In the final step of DTSA, when the algorithm attempts to cover a branch, it may need to try all of the path conditions that apply, performing the $DelNeg$ operation on a specific predicate several times and calling the solver to solve each modified path condition. This process may ultimately lead

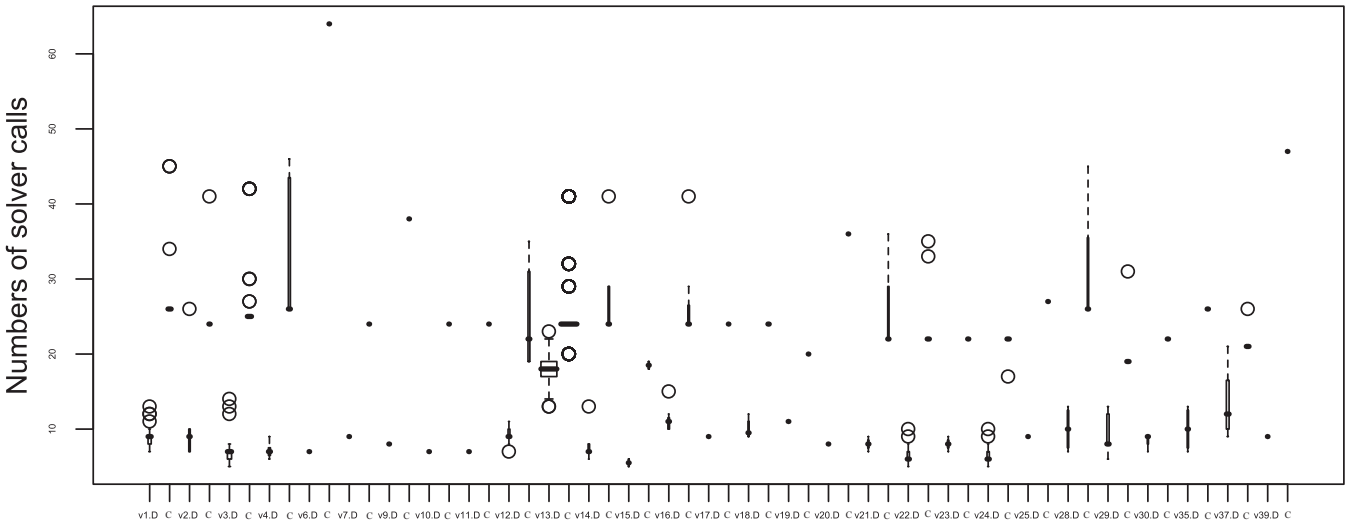

Figure 2. Solver calls: DTSA vs Concolic

Table I
DIFFERENCES IN NUMBERS OF SOLVER CALLS

Version	Mean difference	ρ-value	Version	Mean difference	ρ-value	Version	Mean difference	ρ-value
v1	-19.42	< 0.0001	v13	-7.57	< 0.0001	v23	-14.00	< 0.0001
v2	-15.60	< 0.0001	v14	-19.67	0.001	v24	-14.92	< 0.0001
v3	-20.46	< 0.0001	v15	-	-	v25	-	-
v4	-25.64	< 0.0001	v16	-15.86	0.001	v28	-20.75	0.034
v6	-	-	v17	-	-	v29	-10.54	< 0.0001
v7	-	-	v18	-14.00	< 0.0001	v30	-13.50	< 0.0001
v9	-	-	v19	-	-	v35	-16.00	0.002
v10	-	-	v20	-	-	v37	-7.95	< 0.0001
v11	-	-	v21	-17.50	0.02	v39	-	-
v12	-15.22	< 0.0001	v22	-17.53	< 0.0001	total	-10.47	< 0.0001

Table II
COVERAGE RESULTS

	Branches missed by Concolic	Branches missed by DTSA		Branches missed by Concolic	Branches missed by DTSA		Branches missed by Concolic	Branches missed by DTSA
v1	2.96	0	v13	2.89	0	v23	12	3.00
v2	2.00	2.60	v14	2.83	2.83	v24	3.08	1.00
v3	2.87	2.00	v15	3.00	0	v25	3.00	0
v4	2.91	0	v16	2.86	0	v28	3.00	0
v6	3.00	2.00	v17	3.00	0	v29	2.77	0
v7	3.00	0	v18	3.00	0	v30	2.75	0
v9	6.00	2.00	v19	2.00	0	v35	3.00	0
v10	5.00	5.00	v20	6.00	3.00	v37	2.95	0
v11	6.00	5.00	v21	10.50	3.00	v39	3.00	0
v12	3.00	0	v22	3.15	1.00	total	3.88	1.12

to unnecessary solver calls. We believe that this explains the cases (v13 and v37) in which some runs of DTSA utilized more solver calls than some runs of concolic testing.

As Table II illustrates, an application of full concolic testing is typically less effective than an application of DTSA. This effectiveness gap is particularly strong on versions v21 and v23. We attempted to discern the reasons behind this gap, and we conjecture that the changes to these versions are likely responsible. In both versions a function is replaced by a value that is one of two possible returned values from this function at different positions. The values returned by the function have impacts on subsequent predicates encountered in execution. The changes to return values render it difficult (but not impossible) to cover some branches. However, this difficulty is lessened for DTSA where multiple test cases are available to work from.

The lessons suggested by this analysis are that the modifications made to programs can matter, but also, having multiple inputs (multiple test cases) available that reach code close to changes can facilitate generation of applicable tests. This re-use of prior test cases is not available to an ordinary application of concolic testing "from scratch", and it appears

to make a difference in our ability to generate test cases that focus on modifications.

VI. Conclusions and Future Work

In this paper, we presented a new test suite augmentation technique, DTSA, that combines an existing RTS technique with a modified concolic testing approach to generate test cases that reach code that has not been covered by old test cases. The results of our empirical study provide evidence that DTSA can be effective and efficient.

Our current DTSA prototype has several limitations; the algorithm behind our implementation, however, is not thus restricted, and so we plan to extend our implementation to make it more scalable and to operate on larger and more complex systems. In this way we will be able to expand the scope of our empirical studies and examine the extent to which the results reported in this paper generalize. We also intend to consider other approaches to test generation.

Acknowledgments

This work was supported in part by NSF under Award CNS-0454203 to the University of Nebraska - Lincoln. We thank Wayne Motycka for many helpful suggestions.

References

[1] H. Leung and L. White, "Insights into regression testing," in *Proc. Conf. Softw. Maint.*, Oct. 1989, pp. 60–69.

[2] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Comm. ACM*, vol. 41, no. 5, pp. 81–86, May 1998.

[3] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Meth.*, vol. 6, no. 2, pp. 173–210, Apr. 1997.

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

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

[6] D. Binkley, "Semantics guided regression test cost reduction," *IEEE Trans. Softw. Eng.*, vol. 23, no. 8, pp. 498–516, Aug. 1997.

[7] R. Gupta, M. Harrold, and M. Soffa, "Program slicing-based regression testing techniques," *J. of Softw. Test., Verif., Rel.*, vol. 6, no. 2, pp. 83–111, Jun. 1996.

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

[9] E. Díaz, J. Tuya, R. Blanco, and J. Javier Dolado, "A tabu search algorithm for structural software testing," *Comp. Op. Res.*, vol. 35, no. 10, pp. 3052–3072, 2008.

[10] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Softw. Test., Verif. Rel.*, vol. 9, pp. 263–282, Sep. 1999.

[11] H. Waeselynck, P. Thévenod-Fosse, and O. Abdellatif-Kaddour, "Simulated annealing applied to test generation: Landscape characterization and stopping criteria," *Emp. Softw. Eng.*, vol. 12, no. 1, pp. 35–63, 2007.

[12] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. Int'l Symp. Found. Softw. Eng.*, Sept. 2005, pp. 263–272. [Online]. Available: http://dx.doi.org/10.1145/1081706.1081750

[13] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 529–551, Aug. 1996.

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

[15] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *Proc. Conf. Comp. Comm. Sec.*, Oct 2006, pp. 322–335. [Online]. Available: http://dx.doi.org/10.1145/1180405.1180445

[16] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. Conf. Prog. Lang. Des. Impl.*, June 2005, pp. 213–223. [Online]. Available: http://dx.doi.org/10.1145/1065010.1065036

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

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

[19] A. Kinneer, M. Dwyer, and G. Rothermel, "Sofya: A flexible framework for development of dynamic program analysis for Java software," University of Nebraska - Lincoln, Tech. Rep. TR-UNL-CSE-2006-0006, Apr. 2006.

[20] R. Vallée-Rai, "Soot: A Java Bytecode Optimization Framework," Master's thesis, McGill University, 2000.

[21] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Emp. Softw. Eng.: Int'l J.*, vol. 10, no. 4, pp. 405–435, 2005.

[22] B. Dutertre and L. de Moura, "The Yices SMT solver," http://yices.csl.sri.com/tool-paper.pdf, SRI International, Aug 2006.