CarFast: Achieving Higher Statement Coverage Faster

Sangmin Park
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
sangminp@cc.gatech.edu

Ishtiaque Hussain, Christoph Csallner
University of Texas at Arlington
Arlington, TX 76019, USA
ishtiaque.hussain@mavs.uta.edu, csallner@uta.edu

Kunal Taneja
Accenture Technology Labs and North Carolina State University
Raleigh, NC 27606, USA
ktaneja@ncsu.edu

B. M. Mainul Hossain
University of Illinois at Chicago
Chicago, IL 60607, USA
bhossa2@uic.edu

Mark Grechanik
University of Illinois and Accenture Technology Lab
Chicago, IL 60601, USA
drmark@uic.edu

Chen Fu, Qing Xie
Accenture Technology Labs
San Jose, CA 95113, USA
{chen.fu, qing.xie}@accenture.com

ABSTRACT

Test coverage is an important metric of software quality, since it indicates thoroughness of testing. In industry, test coverage is often measured as statement coverage. A fundamental problem of software testing is how to achieve *higher* statement coverage *faster*, and it is a difficult problem since it requires testers to cleverly find input data that can steer execution sooner toward sections of application code that contain more statements.

We created a novel fully automatic approach for *aChieving higher stAtement coveRage FASTer (CarFast)*, which we implemented and evaluated on twelve generated Java applications whose sizes range from 300 LOC to one million LOC. We compared CarFast with several popular test case generation techniques, including pure random, adaptive random, and Directed Automated Random Testing (DART). Our results indicate with strong statistical significance that when execution time is measured in terms of the number of runs of the application on different input test data, CarFast outperforms the evaluated competitive approaches on most subject applications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution, Testing tools*

Keywords

Testing, Statement Coverage, Experimentation

1. INTRODUCTION

Test coverage is an important metric of software quality [63], since it indicates thoroughness of testing. *Statement coverage*, which measures the percentage of the executed statements to the total number of statements in the application under test [63], is viewed as an important kind of test coverage. Achieving higher statement coverage is correlated with the probability of detecting more defects [51, 49, 38, 10] and increasing reliability of software [44, 11]. Even though it is agreed that statement coverage alone may not always be a strong indicator of software quality [36, page 181], it is a general consensus that achieving higher statement coverage is desirable for gaining confidence in software quality, and it serves as an indicator for test completeness and effectiveness [51, 11, 59].

Statement coverage is widely used in industry as a common criterion for thoroughness of software testing. Different standards require achieving high levels of statement coverage: for example, avionics industry standard, `DO-254`, demands that close to 100% statement coverage be achieved, and avionics industry standard, `DO-178B` and automotive industry standard, `IEC 61508` detail different requirements on achieving statement coverage. Many different organizations use statement coverage as the major criterion for measuring the quality of software testing [18, 39, 52, 58, 7]. Given the importance of statement coverage, how common it is, and how widely it is used to evaluate the thoroughness of testing, it is not surprising that statement coverage is the one coverage metric that is supported by almost all coverage-based testing tools, whereas other coverage measures (e.g., branch coverage, method coverage, or class coverage) are supported by fewer tools [62, 60].

1.1 Higher Statement Coverage Faster

Achieving higher statement coverage means that testers have to select test input data with which they can execute larger portions of application code. Higher statement coverage is always better for increasing the confidence of stakeholders in the quality of software; however, 100% statement coverage is rarely achieved, especially when testing large-scale applications [51, 45, 18]. The faster these testers achieve higher coverage, the lower is the cost of testing [35], since testers can concentrate sooner on other aspects of testing with the selected input data, for example, performance and functional testing with oracles.

We measure the speed with which a certain level of statement coverage is achieved both in *the number of test runs* of the application under test (AUT) with different test input data (i.e., iterations of executing the same application with different input data) and in *the elapsed time* of running the AUT. While the elapsed time gives

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

the absolute value of the time it takes to reach a certain level of coverage; measuring the number of iterations, which essentially means number of test cases to achieve the coverage goal, is important for many reasons, e.g., using fewer test cases requires less manual effort for creating test oracles for these test cases.

Moreover, measuring the number of iterations provides an insight into the potential of a given approach. The elapsed time includes time for generating or selecting input data, and time to run the AUT using these data. In many systematic test case generation or selection approaches, the time spent on generating or selecting input data is significant. Thus, if an approach achieves a higher test coverage using fewer iterations, but spends more time on each iteration, this time can eventually be reduced by improving the efficiency of the particular approach.

A big and important challenge is to get higher statement coverage faster for nontrivial applications that have a very large space of input parameter values. Many nontrivial applications have complex logic that programmers express by using different control-flow statements, which are often deeply nested. In addition, these control-flow statements have branch conditions that contain expressions that use different variables whose values are computed using some input parameters. In general, it is difficult to choose specific values of input parameters to direct the execution of these applications to cover specific statements.

The maximum test coverage is achieved if an application is run with all allowed combinations of values for its inputs. Unfortunately, this is often infeasible because of the enormous number of combinations; for example, 20 integer inputs whose values only range from zero to nine already leave us with 10^{20} combinations. Knowing what combinations of test input data to select for different input parameters to drive the AUT towards the statements nested inside various branches is very difficult. Thus, a fundamental problem of software testing is how to achieve higher statement coverage *faster* by selecting specific values of input parameters that lead the executions of applications to cover more statements in a shorter time period.

1.2 A Novel Approach

We created a novel approach for *aChieving higher stAtement coveRage FASTer (CarFast)* using the intuition that higher statement coverage can be achieved faster if input data are selected to drive the execution of the AUT toward branches that contain more statements. That is, if the condition of a control-flow statement is evaluated to `true`, some code is executed in the scope of this statement. The statements that are contained in the executed code are said to be controlled by or contained in the corresponding branch of this control-flow statement. In program analysis, these statements are said to be *control dependent* [48] on the control-flow statement.

In CarFast, static analysis is used to estimate how many statements are contained in each branch. Once it is known what branches contain more statements, CarFast uses a constraint-based selection approach [41] to select input data that will guide the executions of the AUT towards these branches. In CarFast, input data are not generated, but they rather come from external databases as we describe it in Section 2.5. Comparing to CarFast, many automatic test data generation techniques use computationally expensive *constraint solvers* [14, 19, 25, 43] to generate test data for achieving higher test coverage. Using these solvers negatively affects the scalability of these techniques. By replacing constraint solvers with selectors, not only does CarFast achieve higher coverage faster, but also it achieves better scalability (Section 5).

CarFast offers multiple benefits: it is fully automatic since it does not require any intervention by testers; it is directed since it explores branches in the *control-flow graph (CFG)* of the AUT rather than an enormous space of the combinations of the values for the input test data; and it is scalable on large-scale applications with up to 500kLOC as demonstrated by our experiments with twelve Java applications with up to one million LOC. Even though we built CarFast to work with Java programs, there are no fundamental limitations to generalizing it to other languages and platforms, such as C# or C++. This approach may be generalized for other coverages that can be (1) statically approximated; and (2) dynamically computed. To the best of our knowledge, CarFast is the first approach that defines and uses static program analysis-based coverage gain prediction to achieve higher coverage faster, and this work is the first that evaluated this idea with strong statistical significance on a large number of subject applications of varying sizes.

1.3 Our Contributions

This paper makes the following contributions:

- We developed a novel algorithm for achieving higher statement coverage faster and we implemented it as part of CarFast.
- We applied CarFast to twelve Java applications whose sizes range from 300 LOC to one million LOC that we generated using stochastic parse trees [57, 32]. CarFast, subject Java applications, and the stochastic application benchmark generator are available for public use[1].
- We conducted large-scale experiments using Amazon EC2 to evaluate CarFast and competitive approaches against one another, specifically, pure random and adaptive random testing, and Directed Automated Random Testing (DART) [26] using a rigorous experimental evaluation methodology [2]. The results show that when execution time is measured in terms of iterations, CarFast outperforms evaluated competitive approaches for most subject AUTs with strong statistical significance.
- Finally, we compared pure random testing to adaptive random testing to address an open issue in determining which approach is better [1, 47]. Adaptive random performs statistically as good as pure random testing when lower coverage is targeted and time is measured in terms of iterations. When higher coverage is targeted, pure random beats adaptive random testing with strong statistical significance.

2. THE CARFAST APPROACH

In this section, we give an illustrative example of how our approach works, we formulate our hypothesis upon which we designed our approach, and we give an algorithm of CarFast.

2.1 An Illustrative Example

An illustrative example is shown in Figure 1 using Java-like pseudo-code. Line numbers to the right should be thought of as labels, as much code is omitted for space reasons. This example has `if-else` statements that control three branches, where branch labels are shown in comments along with the numbers of statements that these branches control. These numbers are given purely for an illustrative purpose.

Consider executing this code with randomly selected input values $i1 = 3$ and $i2 = 7$, which leads to the `else` branch `3` in lines `5–8`. The number of distinct uncovered statements that are reachable from this branch is 100, which is significantly less than the

[1]All tools and subject applications are available for download at: http://www.carfast.org

```
if( i1 == 10) {                                  1
  .. // branch 1: 300 statements                 2
} else if( i2 == 50 ) {                          3
  .. // branch 2: 600 statements                 4
} else {                                         5
  .. // branch 3: 100 statements                 6
  if(..) { if(..) { .. } .. }                    7
}                                                8
```

Figure 1: An illustrative example.

numbers of statements contained in the two other branches. We say that a previously uncovered statement `S` is contained in a branch `B`, if there exists a concrete input that triggers an execution that evaluates the condition of the branch `B` and covers `S`.

Besides the lower number of statements reachable from branch 3, the control flow within branch 3 is also more complex than the control flow in branch 1 or branch 2 (as denoted by a few example nested `if-else` statements). Clearly, this is one of the worst test inputs since it covers only 10% of this code at best. To achieve higher coverage faster, we would like to learn from this execution to select a test input that satisfies $i1 \neq 10 \wedge i2 = 50$ to steer the next run toward branch `2`, since it contains the biggest number of distinct reachable uncovered statements (i.e., 600), thus increasing test coverage up to 70%.

However, none of the existing approaches can systematically steer the execution towards branch `2`, since it is the nature of randomization to select data points independently from one another from the input space. DART dynamically analyzes program behavior using random testing and generates new test inputs automatically to direct the execution systematically along alternative program paths [26]. A problem is that the original DART algorithm will keep exploring all nested branches in branch `3` using a *depth-first search* algorithm. That is, DART keeps exploring the branch for a while even though the gain in coverage will rather be minuscule.

2.2 Our Observation, Preliminary Study, and Hypothesis

We observe that many applications have a few branches that contain large bodies of statements and many more branches that contain only few statements. This observation is supported by a preliminary study we did on three nontrivial and widely used open-source Apache applications, `log4j`[2], `Ant`[3], and `JMeter`[4]. For each application, we counted the number of statements in each basic block. Our results indicate that the number of statements per basic block approximates the power law [42], i.e., approximately 80% of the statements are in 20% of the basic blocks. Specifically, 20% of the basic blocks contain 73% of the statements in `Ant`, 65% in `Jmeter`, and 66% in `log4j`. Assuming that this observation holds for many applications, our intuition is that we can exploit this skewed size distribution of the basic blocks in test case selection. I.e., we want to systematically steer AUT execution towards these large basic blocks, to achieve higher statement coverage faster.

We hypothesize that we can significantly accelerate test selection techniques by guiding the selection to test input data that covers those branches that contain more uncovered statements, by estimating the number of distinct statements. This hypothesis is rooted in the essence of systematic testing, which is a counterpart to random testing where test data inputs are selected without any prior knowledge [30]. To test our hypothesis, we combine systematic and random testing approaches in a novel way using the insight that higher coverage can be achieved faster if it is known which distinct branches that are still uncovered contain more statements. Knowing the constraints from branch conditions may enable selection of test input data that leads to execute the statements within those branches.

[2]Version 1.2.16, http://logging.apache.org/log4j

[3]Version 1.8.2, http://ant.apache.org

[4]Version 1.0, http://jmeter.apache.org

2.3 CarFast by Example

We review how our approach works using our illustrative example of Figure 1. Once branch `3` of this code is executed using input values $i1 = 3$ and $i2 = 7$, constraints $C_1 : i1 \neq 10$ and $C_2 : i2 \neq 50$ can be learned automatically. Since branch `2` contains the biggest number of statements (i.e., the 600 statements of line `4`), constraint C_2 can be negated and the resulting constraint formula will be $C_1 \wedge \neg C_2$ or $i1 \neq 10 \wedge i2 = 50$. In the next step, test input data is obtained that fits this constraint, that is the value of $i1 \neq 10$ and $i2 = 50$. This process can be repeated as often as necessary to achieve higher coverage.

In the worst case, this approach may result in test input data that leads execution towards already executed branches or branches that have fewer statements. For example, if the first input data is selected $i1 = 10$ and branch `1` is executed, no additional useful constraint except for $i1 \neq 10$ will be learned during this step to help our approach to steer execution toward branch `2`. However, random selection allows testers to select completely different data, which in turn will lead to different execution profiles and learning more constraints [30, 29, 3]. Our hypothesis is that by learning more of these constraints with each execution of the AUT, it is possible to converge to higher coverage faster. Verifying this hypothesis is a goal of this paper.

2.4 Ranking Branches by Statements

To understand which branches affect statement coverage the most, we rank each branch (if, loop, etc.) by the number of statements it contains (the number of statements that are transitively control-dependent on that branch). Executing higher ranked branches enables achieving higher statement coverage faster. To rank branches, we construct and traverse a CFG of the AUT, then we count the (inter-procedural) statements that are control-dependent on each branch condition. I.e., if in method `c`, branch `b` transitively controls a call to method `m`, then the statements of `m` are treated as if they were in-lined into the calling method `c`, and some statements of `m` (and possibly statements of methods called by `m`) may be included in the count of statements that are control-dependent on `b`.

Specifically, if the method `m` is virtual, we perform virtual call resolution using static class hierarchy analysis, and count all the statements in all the target methods; but use the one with the maximum statements to determine the number of statements controlled by branch `b`.

2.5 Selecting Existing Test Input Data

We assume that the test input data come from existing repositories or databases. This is a common practice in industry, we confirmed it after interviewing professionals at IBM, Accenture, two large health insurance companies, a biopharmaceutical company, two large supermarket chains, and three major banks. For instance, the *Renters Insurance Program* designed and built by a major insurance company has a database that contains approximately 78 million customer profiles, which are used as the test input data.

As part of CarFast, we translate constraints into SQL queries against such existing databases [41]. For example, to find values

for input variables $i1$ and $i2$ that satisfy the constraint $i1 \neq 10 \wedge i2 = 50$, the following SQL query is executed: `SELECT * FROM InputTbl WHERE i1 != 10 AND i2 = 50`. Some of these SQL queries include millions of conditions in the `WHERE` clause, which caused runtime problems in some commercial strength database management system such as Microsoft SQL server. To scale CarFast, we developed a lightweight and efficient SQL-based constraint evaluator that we describe in Section 3.3.

2.6 The Algorithm

The algorithm `CarFast` is shown in Algorithm 1. This algorithm takes as its input the set of the input parameter values T, the AUT P, and the set of accounted AUT branches, B. The total coverage score $totalCov$ is computed and returned in line `31` of the algorithm.

In step `2`, the algorithm initializes the values for total coverage $totalCov$ to zero, the set of covered branches B_{cov} whose statements are covered, and the set of constraints to the empty set. In step `3`, the procedure `ComputeBranchFun` is called that computes the function *BRank* that maps each branch of the AUT, P, to the approximate number of statements that are reachable from this branch. Next, in step `4`, the procedure `Sort` sorts elements of input set B in the descending order according to the number of statements using the function *BRank*, producing the sorted set B_s. In step `5`, the procedure `GetRandomTestInput` randomly selects a data object, t from the set of the input parameter values, that is, Input Test Data, T, and this data object is removed from the set in step `6`. This algorithm runs the loop between steps `7`–`30`, which terminates on the condition of the reached time limit or desired coverage (i.e., the predefined value *covCeiling*). In step `8`, the AUT, P is executed using the input data t, resulting in the updated value of $totalCov$, added branches that were covered during this execution to the set of covered branches, B_{cov}, and added constraints that are learned during this execution. Then, in the `for` loop in steps `10`–`25`, each member branch in the set B_s is examined to check whether it was covered in the previous run of the AUT. If some branch, b_k was covered, it is removed from the set B_s in line `23`, otherwise, the first occurrence of the corresponding constraint, C_k is inverted (ignoring the following constraints in the path condition) in line `12`.

By treating a constraint as a query to obtain input data that satisfy the conditional WHERE clause, the subsets of test input data, $\{t_c\}$ is obtained in line `13` that satisfy this clause, that is the flipped constraint. If $\{t_c\}$ is empty, then no test input data from our input database can lead to the desired branch, and this message is issued in line `20`. Otherwise, one input, t is randomly selected from the set $\{t_c\}$ in line `15` and the control is returned to line `25` and eventually to line `8`, where the AUT is run with this input thereby repeating the loop.

In some cases, it may not be possible to know the exact constraints to reach certain statements, since the set of constraints that is collected by the concolic engine corresponds to reaching different nodes of the CFG, and subsequently different statements. Flipping these constraints and solving these flipped constraints may result in input data that will lead the AUT toward other uncovered statements, but not necessarily the desired statements. However, as more constraints are collected with newly obtained test input data, these constraints eventually enable CarFast to narrow down the scope of the executed statements to the ones that are desirable.

This works only if the input test data set contains an input that can reach such statements. If the application contains a statement that is not reachable with the inputs from Input Test Data, then CarFast will never cover that statement. However, as the results of the experimental evaluation show in Section 5, CarFast outperforms other competitive approaches under different conditions.

Algorithm 1 The CarFast algorithm.

1: **CarFast**(TestInputData T, AUT P, AUT Branches B)
2: $totalCov \leftarrow 0, B_{cov} \leftarrow \{\emptyset\}, C \leftarrow \emptyset$ {Initialize values of the total statement coverage, the set of covered branches, and the set of constraints.}
3: ComputeBranchFun(P) $\mapsto BRank : \{B\} \ni b \rightarrow rank$
4: Sort($B, BRank$) $\mapsto B_s$ {Sort elements of the set in the descending order by their rank using the function $BRank$}
5: GetRandomTestInput(T) $\mapsto t \in T$
6: $T \mapsto T \setminus t$
7: **repeat**
8: RunAUT(P,t) $\mapsto [(totalCov \mapsto totalCov + cov'), (B_{cov} \mapsto B_{cov} \cup \Delta_B), (C \mapsto C \cup (C_1 \wedge \ldots \wedge C_n))]$
9: foundTest4Branch $\leftarrow$ **false**
10: **for all** $b_k \in B_s$ **do**
11: **if** $b_k \notin B_{cov}$ **and** $C_k \in C$ **then**
12: FlipConstraint(C) $\mapsto (C \mapsto C_1 \wedge \ldots \wedge \neg C_k))$
13: GetTestInput(C) $\mapsto \{t_c\} \in T$
14: **if** $\{t_c\} \neq \emptyset$ **then**
15: GetRandomTestInput($\{t_c\}$) $\mapsto t$
16: $T \mapsto T \setminus t$
17: foundTest4Branch $\leftarrow$ **true**
18: break the **for** loop
19: **else**
20: **print** Given constraint C cannot be satisfied
21: **end if**
22: **else**
23: $B_s \mapsto B_s \setminus b_k$
24: **end if**
25: **end for**
26: **if** foundTest4Branch = **false then**
27: GetRandomTestInput(T) $\mapsto t$
28: $T \mapsto T \setminus t$
29: **end if**
30: **until** time limit is not reached or $totalCov < covCeiling$
31: **return** $totalCov$

3. IMPLEMENTATION AND DEPLOYMENT

In this section, we describe main challenges and the salient features of our implementation and deployment including the concolic engine.

3.1 Main Challenges

Our implementation goal is to demonstrate that CarFast is viable by applying it to large-scale AUTs. There are two main challenges: it is memory-intensive and it contains CPU-intensive components. Extracting constraints by executing AUTs takes time and most importantly, significant amounts of memory. Typically, concolic engines incur more than an order of magnitude overhead from normal program execution. Memory footprint of concolic engines increases quickly as the engines must keep track of symbolic representations of all aspects of the current program execution (e.g., symbolic representations of the static fields of all loaded classes) as execution traces get longer. In our experiments, one extracted constraint from a 50KLOC AUT is over five megabytes and its size grows to 50GB for one million LOC AUT! Moreover, solving such constraints can take a long period of time, since it involves performing queries on large sets of input test data.

3.2 Concolic Execution Engine

We used Dsc [33], a Java dynamic symbolic execution engine (i.e., a concolic engine) for Java AUTs in CarFast. Below we describe its main features and how we adapted it to scale to large AUTs.

3.2.1 Overview of Dsc

Dsc instruments the bytecode of AUTs automatically by inserting method calls (i.e., callbacks) after each instruction in the code. During AUT execution, the callbacks enable Dsc to maintain the symbolic state by mirroring the effects of each user program instruction, including the effects of reading and writing heap (i.e., array and object field) locations, performing integer and floating point arithmetic, following the local and inter-procedural control-flows, and handling exceptions.

Dsc integrates well with the existing Java execution environments; it does not require any modifications of the user application code, or the virtual machine. Dsc uses the instrumentation facilities provided by the JVM of Java 5 to instrument the user program at load-time [15], using the open source bytecode instrumentation framework ASM [8]. By manipulating programs at the bytecode level, Dsc extends its analysis from the user code into all libraries called by these programs. In addition, Dsc allows users to selectively exclude classes from instrumentation.

3.2.2 The Dumper Mode of Dsc

Dsc in its normal mode represents every concrete computation by a corresponding symbolic expression, caches it in memory and utilizes it later when the same computation is repeated or is used in a subexpression. However, nontrivial applications contain large number of computation steps and caching symbolic expressions quickly exhaust the available heap memory. Moreover, often computations are done in loops or recursive call chains and when concolic engines process these loops or recursive call chains, they produce long symbolic expressions, which are in turn used in subsequent computations adding quickly to the total length of the resulting symbolic expression. As a result, even for moderate size programs, concolic executions quickly exhaust all memory.

To scale to large applications, we introduced a dumper mode for Dsc to minimize the memory consumption for the symbolic state representation. Instead of caching symbolic expressions in heap memory, this dumper mode introduces local variables (symbols) for each expression and *dumps* or writes these expressions to the disk. Later, a dynamic lookup and replacement technique is used on the *dump* file to build the constraints or path condition involving input parameters.

3.3 Constraint-Based Selector

To improve the scalability of CarFast, we developed a constraint-based selector rather than using off-the-shelf constraint solvers. Our motivation is twofold: improving the speed of computation and better utilizing resources. Specifically, our concolic engine, Dsc is a 32-bit tool, meaning that it can only use less than four gigabytes of RAM at a time. Adding a constraint solver to the process space of Dsc would significantly reduce available memory. To address this problem, we implemented the constraint-based selector as a separate server process that can serve many Dsc clients simultaneously through socket interfaces.

For the implementation of the constraint solver, at the beginning, we kept all the data in a relational database. Then, by running a query, we tried to select the data that satisfy the constraint given as the WHERE clause of the query. But, the traditional RDBMSs failed to process a query with such a big constraint part in the WHERE clause.

As a different approach, we wrote a set of production rules to define a formal grammar so that every possible constraint can be recognized by that grammar. We used the ANTLR tool[5] for processing the grammar and its languages. With the help of ANTLR, we were able to parse and process much larger constraints.

When the server process (constraint solver) receives a query from a client, the server builds the abstract syntax tree for the condition part (WHERE clause) of the query. Then, it evaluates the tree, in a bottom-up fashion, against all possible input data in the repository. Finally, as test input data, the server process returns values for the parameters that satisfy all the conditions in the condition-part of the query.

3.4 Miscellaneous

We implemented branch ranking using Java static analysis and transformation engine called Soot [54]. All conditional branch statements in the AUTs are ranked using the approach described in Section 2.4. At runtime, we used EMMA[6] to compute and report statement coverages. Also, we modified callback functions in Dsc to keep track of the covered branches during the test execution to reset rankings of the already executed branches to zero to avoid repeatedly executing already covered statements.

4. EXPERIMENTS

To determine how effective CarFast is in achieving higher statement coverage faster, we conducted an experiment with competitive approaches such as random testing, adaptive random testing, and DART on twelve Java applications (i.e., AUTs) whose sizes range from 300 LOC to one million LOC. In this section, we briefly describe these competitive approaches, provide the methodology of our experimental design, explain our choice of subject AUTs, and discuss threats to validity.

4.1 Variables

Main independent variables are the subject AUTs, the value of test coverage that should be achieved for AUTs in each experiment, and approaches with which we experiment (i.e., random, adaptive random testing, DART, and CarFast). A dependent variable is the execution time that it takes to achieve a given test coverage. We measure the execution time both in terms of elapsed time, E and as a number of iterations of AUT executions with different input values, I. The effects of other variables (the structure of AUT and the types and semantics of input parameters) are minimized by the design of this experiment.

4.1.1 Random Testing

Random testing approach, as the name suggests, involves random selection of test input data for input parameter values, and in that it showed remarkably effective and efficient for exploratory testing and bug finding [4, 24]. A seemingly "stupid" idea of random testing proved often more effective than systematic sophisticated testing approaches [30, 29]. To prove our claims in this paper, our goal is to show under what conditions CarFast outperforms random testing with strong statistical significance.

4.1.2 Adaptive Random Testing

Adaptive random testing (ART) is a controversial refinement of the baseline random testing where randomly selected data are dis-

[5]http://www.antlr.org/

[6]http://emma.sourceforge.net

tributed evenly across the input data space [12]. In that, ART introduces a certain level of control over how input data is selected when compared with the baseline random testing. A recent implementation of ART for object-oriented languages is ARTOO, which we use as a competitive approach to CarFast in our experiments [13]. Prior to our experiment, ARTOO was evaluated on eight classes from the EiffelBase library, and the sizes of these classes ranged from 779LOC to 2,980LOC. Recently, Arcuri and Briand presented statistically significant results of experiments that question the effectiveness of ARTOO with respect to bug detection for programs with seeded faults [1]. Meyer pointed out in his response [47] that the programs with seeded faults behave much differently from programs with real faults. Moreover, Arcuri and Briand measured the time to find the first fault as a testing metric, which may not be a rigorous metric [50]. Therefore, we performed an experiment comparing random testing and ARTOO with a set of small to large programs with statement coverage as a testing metric. In this paper, we also address a research question of how effective ARTOO is in achieving higher coverage faster against competitive approaches including random testing.

4.1.3 DART

Directed Automated Random Testing (DART) is an approach that uses a concolic engine to generate test inputs that explore different execution paths of a program [26]. In the original DART algorithm, path exploration is conducted in *Depth-First-Order (DFO)* or *Breath-First-Order (BFO)* of navigating the CFG of the AUT. We faithfully re-implemented DART using Dsc, so that we can evaluate it in an unbiased fashion against CarFast. In the original paper [26], DART was previously evaluated only on three C applications whose sizes range from a dozen LOC to 30kLOC. Even though there are many implemented variations of DART, (e.g., jCUTE, KLEE, Pex), DART has never been evaluated with strong statistical significance on benchmark AUTs.

4.2 Methodology

Our goal is to determine which approach achieves higher statement coverage faster. Given the complexity of the subject AUTs, it is not clear what is the highest coverage that can be achieved for these AUTs, and given a large space of input data, it is not feasible to run the AUTs on all inputs to obtain the highest statement coverage. These limitations dictate the methodology of our experimental design, specifically for choosing the threshold for the desired test coverage, which is AUT-specific and in general less than 100% for a number of reasons, not the least of which is the presence of unreachable code in AUTs. Before conducting experiments, we run each benchmark AUT against pairwise test input data, and use the resulting achieved coverage as the coverage threshold for it. Each experiment run is halted when either it hits the coverage threshold, or the execution time limit (24 hours) is reached. The time limit is determined experimentally. See Section 5.2 for details.

We aligned our methodology with the guidelines for statistical tests to assess randomized algorithms in software engineering [2]. Our goal is to collect highly representative samples of data when applying different approaches, perform statistical tests on these samples, and draw conclusions from these tests. Since our experiments involve random selection of input data, it is important to conduct the experiments multiple times to pick the average to avoid skewed results. For each subject application, we ran each experiment 30 times with each approach on the same AUT to consider collected data a good representative sample. It means that for a total of 12 AUTs we ran 30 experiments for each of the four approaches, resulting in a total of $12 \times 4 \times 30 = 1,440$ experiment runs.

To evaluate our hypotheses, we ran statistical tests based on the assumption that the population is normally distributed. The law of large numbers states that if the population sample is sufficiently large (between 30 to 50 samples), then the central limit theorem applies even if the population is not normally distributed [56, page 244-245]. Since we have 30 sample runs for each AUT for each configuration, the central limit theorem applies, and the above-mentioned tests have statistical significance.

Experiments are carried out in Amazon EC2[7] virtual machine *large* instances with the following configuration: 7.5 GB RAM, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 35 GB instance storage. We set a 24-hour time limit for each experiment run. So the estimated total runtime is $1,440 \times 24 = 34,560$ hours. With the cost of USD 0.48 per large instance per hour as of September, 2011, the estimated cost of this experiment was around USD 16,500. However, we underestimated the cost of building, testing and fixing the experiment environment itself, which resulted in a total cost of around USD 30,000.

4.3 Hypotheses

We introduce the following null and alternative hypotheses to evaluate how close the means are for the Es and Is for control and treatment groups. Unless we specify otherwise, CarFast is applied to AUTs in the treatment group, and other competitive approaches are applied to AUTs in the control group. We seek to evaluate the following hypotheses at a 0.05 level of significance.

H_0 The primary null hypothesis is that there is no difference in the values of test coverage that AUTs can achieve in a given time interval.

H_1 An alternative hypothesis to H_0 is that there is statistically significant difference in the values of test coverage that AUTs can achieve in a given time interval.

Once we test the null hypothesis H_0, we are interested in the directionality of means, μ, of the results of control and treatment groups, where S is either I or E. In particular, the studies are designed to examine the following null hypotheses:

H1: CarFast versus Random. The effective null hypothesis is that $\mu_S^{CarFast} = \mu_S^{Rand}$, while the true null hypothesis is that $\mu_S^{CarFast} \geq \mu_S^{Rand}$. Conversely, the alternative hypothesis is $\mu_S^{CarFast} < \mu_S^{Rand}$.

H2: CarFast versus ARTOO. The effective null hypothesis is that $\mu_S^{CarFast} = \mu_S^{ARTOO}$, while the true null hypothesis is that $\mu_S^{CarFast} \geq \mu_S^{ARTOO}$. Conversely, the alternative hypothesis is $\mu_S^{CarFast} < \mu_S^{ARTOO}$.

H3: CarFast versus DART. The effective null hypothesis is that $\mu_S^{CarFast} = \mu_S^{DART}$, while the true null hypothesis is that $\mu_S^{CarFast} \geq \mu_S^{DART}$. Conversely, the alternative hypothesis is $\mu_S^{CarFast} < \mu_S^{DART}$.

H4: ARTOO versus Random. The effective null hypothesis is that $\mu_S^{ARTOO} = \mu_S^{Rand}$, while the true null hypothesis is that $\mu_S^{ARTOO} \leq \mu_S^{Rand}$. Conversely, the alternative hypothesis is $\mu_S^{ARTOO} > \mu_S^{Rand}$.

[7] http://aws.amazon.com/ec2/instance-types as of March 10, 2012

The rationale behind the alternative hypotheses to `H1`, `H2`, and `H3` is that Carfast achieves certain test coverage faster than other approaches. The rationale behind the alternative hypothesis to `H4` is that the random approach outperforms ARTOO as suggested by Arcuri and Briand [1].

4.4 Input Test Data Repository

Recall that instead of generating test data, CarFast selects test input data from existing repositories. Most nontrivial applications have enormous spaces of test input data objects that are constructed by combining values of different input parameters. Even though it is infeasible to create a test data repository that contains the entire input space, it is possible to create combinations of values that will result in a smaller space of input data objects using combinatorial design algorithms, which are frequently used by testing practitioners [28, 16, 40]. Most prominent are algorithms for t–wise combinatorial testing, which requires every possible combination of interesting values of t parameters be included in some test case in the test suite [28]. Pairwise testing is when $t = 2$, and every unique pair of values for each pair of input parameters is included in at least one test case in the test suite. To construct a test data repository for evaluating CarFast, we used the ACTS[8] tool (previously known as FireEye) to generate data for our experiments using pairwise testing from the range of input data $[-50, 50]$ that was chosen experimentally. Since pairwise selection significantly reduces the number of test input data, we added up to one million combinations of test input data values using an unbiased random selection.

4.5 Subject AUTs

Given that we claim significant improvements in CarFast when compared with competitive approaches, it is important to select application benchmarks that are not biased, nontrivial, and enable reproducibility of results among other things. In general, a benchmark is a point of reference from which measurements can be made in order to evaluate and predict the performance of hardware or software or both [46]. Benchmarks are very important for evaluating program analysis and testing algorithms and tools [5, 6, 20, 53].

4.5.1 Challenges With Benchmark Applications

Different benchmarks exist to evaluate different aspects such as how scalable program analysis and testing tools are, how fast they can reach high test coverage, and how effective these tools are in executing applications symbolically or concolically. Currently, a strong preference is towards selecting benchmarks that have much richer code complexity (e.g., nested `if-then-else` statements), class structures, and class hierarchies [5, 6]. Unfortunately, complex benchmark applications are very costly to develop [37, page 3], and it is equally difficult to find real-world applications of wide variety of sizes and software metrics that can serve as unbiased benchmarks for evaluating program analysis and testing approaches.

Consider our situation where different test input data selection and generation approaches are evaluated to determine which approach enables users to achieve higher statement coverage faster. On one extreme, "real-world" applications of low complexity with very few control-flow statements are poor candidate benchmarks, since most test input data generation approaches will perform very well, especially if AUTs take as input parameters only primitive types. On the other extreme, it may take significant effort to adjust these approaches to work with a real-world distributed application whose components are written in different languages and run on different platforms. In addition, current limitations of concolic engines (e.g., manipulating arrays, different types) make it very difficult to select nontrivial application benchmarks to satisfy these limitations. Ideally, a large number of different benchmark applications are required with different levels of code complexity to appropriately evaluate test input data generation tools.

[8] http://csrc.nist.gov/groups/SNS/acts/index.html

A	kLOC	Cl	Meth	NBD	MCC	WMC
1	0.3	4	3	2.5/6	6.3/20	23.5/36
2	0.6	5	5	2.3/5	10.4/30	33.4/56
3	1.2	14	19	2.0/5	6.9/26	24.2/44
4	1.3	18	61	2.2/9	3.8/14	22.4/47
5	2.1	24	49	2.0/5	4.5/13	24.5/36
6	5.2	37	184	2.0/8	5.2/23	42.9/86
7	7.8	38	469	2.2/8	4.3/19	63.4/137
8	24.2	111	765	2.4/8	4.7/23	66.4/102
9	46.7	61	428	4.2/12	22.3/56	249.2/347
10	98.4	96	1,576	3.4/8	10.7/27	325.3/447
11	470.8	311	2,244	4/7	34.1/93	464.3/640
12	1,157.2	781	17,449	3.8/13	13/47	486/631

Table 1: Subject AUT (A) characteristics. Cl = #classes (NOC), Meth = #methods (NOM), NBD = nested block depth, MCC = McCabe cyclomatic complexity, WMC = weighted methods per class. The last three columns show average and maximum values as Avg/Max.

One way to address the problem is to write benchmark applications that satisfy the requirements. However, writing benchmark application from scratch is laborious, not to mention that a significant bias and human error can be introduced [34]. In addition, selecting commercial applications as benchmarks negatively affects reproducibility of results, which is a cornerstone of the scientific method [55, 23], since commercial benchmarks cannot be easily shared among organizations and companies for legal reasons and trade-secret protection. For example, Accenture Human Resource Policy item 69 states that source code constitutes confidential information, and other companies have similar policies. Finally, In addition, more than one benchmark is often required to determine the sensitivity of program analysis and testing approaches based on the variability of results for applications that have different properties [2], making it a very laborious exercise.

Ideally, users should be able to easily generate benchmark applications with desired properties that are similar to real-world applications. This idea has been already successfully used in testing relational database engines, where complex *Structured Query Language (SQL)* statements are generated using a random SQL statement generator [57]. Suppose that a claim is made that a relational database engine performs better at certain aspects of SQL optimization than some other engine. The best way to evaluate this claim is to create complex SQL statements as benchmarks for this evaluation in a way that these statements stress properties that are specific to these aspects of SQL optimization. Since the meaning of SQL statements does not matter for performance evaluation, this generator creates semantically meaningless but syntactically correct SQL statements thereby enabling users to automatically create low-cost benchmarks with reduced bias. In addition, synthetic programs and data have been used widely in computer vision and image processing [22, 31].

4.5.2 Random Benchmark Applications

We define a random program by construction. Every program is an instance of the grammar of the language in which this program is written. We use the grammar to generate branches of a parse tree for different production rules, where each rule is assigned the

probability with which it is instantiated in a program. Starting with the top production rules of the grammar, each nonterminal is recursively replaced with its corresponding production rule. Terminals are replaced with randomly generated identifiers and values, and they are used in expression with certain probability distributions, leading to a syntactically correct program. This approach is widely used in natural language processing, speech recognition, information retrieval [17], and also in generating SQL statements for testing database engines [57].

4.5.3 Subject AUTs For Experimentation

We generated twelve subject AUTs whose sizes range from 303 LOC to over one million LOC using our program generator [32]. To minimize the effect of using different libraries and data types on our experimental design, we allowed only integer data types and standard Java language constructs. Each AUT takes 15 input parameters, this number is chosen experimentally. Table 1 contains characteristics of the subject programs, with the first column showing the names followed by other columns with different characteristics of these AUTs as specified in the caption. We used the generated programs without any tweak, and they are available at the website given in Footnote 1.

4.6 Threats to Validity

The main threat for our experimental design is the selection of subject AUTs and their characteristics. Due to limitations of concolic engines, which requires finite number of values for each input, we had to synthesize the subject AUTs, and these AUTs have high cyclomatic complexity, which makes it difficult to choose values for input parameters to achieve high coverage faster. The results may vary for AUTs that have very simple logic or different source code structures.

The other threat to validity comes from evaluating approaches based on statement coverage rather than using some fault detection metric. In general, even though a connection exists between statement coverage and fault detection capability, the latter is a more robust metric since it goes into the heart of a main goal of testing – bug detection. However, existing approaches for applying fault detection metric use generated mutants, which are not always equivalent to applications with bugs that are introduced by programmers [50]. Finally, statement coverage is also an important metric for stakeholders to obtain confidence from testing applications, and we evaluate this important testing metric.

Finally, a threat to validity is our method for selecting ranges of input data. Since our AUTs are generated, it is unclear what ranges of input values should be chosen and how the number of combinations of input values can be minimized effectively. In our experimental design we used standard practices used by test engineers at different Fortune 500 companies, specifically to apply combinatorial pairwise testing to create sufficiently diverse sets of input test data.

5. RESULTS

In this section, we provide and explain results of experiments and statistical tests to address our hypotheses.

5.1 Testing the Null Hypothesis

We used ANOVA to evaluate the null hypothesis H_0 that the variation in an experiment is no greater than that due to normal variation of individuals' characteristics and error in their measurement. The results of ANOVA confirm that there are large differences between the approaches for coverage for both measures of execution time. As the result shows, all p-values are less than 0.05. Hence, we reject the null hypothesis H_0 and accept the alternative hypothesis H_1.

Statistical results for execution times are shown in Table 2. DART ran out of memory for AUTs A6–A12 and CarFast ran out of memory for AUT A12 (over 1Mil LOC). Based on t-tests for paired two sample for means for two-tail distribution, we reject hypotheses `H1`–`H2` for time measured as iterations, and we reject hypotheses `H3`–`H4` for both measures of time, i.e., iterations and elapsed time. We can summarize these results as following.

- When only iterations are counted, CarFast achieves higher statement coverage faster when compared with the random and ARTOO approaches for all AUTs with strong statistical significance. CarFast also outperforms DART for all AUTs but A5, since both approaches reach the desired coverage in one iteration for A5.

- When only elapsed execution times are counted, random and ARTOO achieve higher statement coverage faster when compared with CarFast for all AUTs, with strong statistical significance. However, when comparing CarFast with DART, CarFast outperforms DART for all AUTs but A5 for the same reason mentioned above.

- When only iterations are counted, the random approach achieves higher statement coverage faster when compared with the ARTOO approach for all AUTs with strong statistical significance.

- When only elapsed execution times are counted, the random approach achieves higher statement coverage faster when compared with the ARTOO approach for all AUTs with strong statistical significance. But when lower coverage is specified, for small programs (e.g., A1, A2, A4 and A5) the random approach does not outperform ARTOO.

5.2 Investigation of Corner Cases

We investigated the corner cases. First, we found that in A5 the maximum coverage, 46%, is achieved in one iteration using all approaches. Since the obtained test coverage with DART is significantly lower when compared with other approaches, we run statistical tests with results for DART excluded, so that we verify that the results of testing hypotheses still hold. We chose 78% (i.e., the minimum coverage of all approaches but DART) as the coverage level for which we extracted execution times for these approaches. The results are consistent with our previous conclusion. We had similar treatment for A1–A4, i.e., we excluded the results that we obtained from DART from data analysis, and got consistent results.

Second, we found that the execution time of CarFast increases at a much faster rate when compared to that of random and ARTOO approaches as program size increases. We analyzed the execution time of CarFast for each benchmark AUT, and we found out that the dumper mode of Dsc takes the most of time for large applications when compared with other components of CarFast. For A11, it takes 22.4 minutes on average per one iteration of an execution, and it takes 75% of entire CarFast execution time. For A12, Dsc ran out of memory. That is, the dumper mode of Dsc should be improved to make CarFast scalable to large programs.

Finally, we found that the accumulated test coverage for CarFast after running for 24 hours is comparable with that of random and ARTOO approaches for A1–A8. As mentioned above, it takes much longer for CarFast to run on larger applications (A9–A12), which results in much fewer iterations (or out of memory) within 24 hours and in achieving less test coverage.

A	*Cov*	App	I_{med}	I_{mean}	I_{min}	I_{max}	SD_I	E_{med}	E_{mean}	E_{min}	E_{max}	SD_E	C_{min}	C_{max}	C_{med}
1	65%	Rand	27	29.6	13	56	12.1	88	97.2	51	174	37.7	86%	88%	88%
		ART	28	28.7	12	62	11.3	83	84.1	36	185	32.7	84%	88%	88%
		DART	504	973.5	126	3911	1003	1410	2784.0	338	11307	2918	65%	81%	74%
		CF	14	13.8	8	19	2.7	495	473.7	183	718	106.8	84%	88%	86%
	84%	Rand	1187	1299.2	282	2859	736	3539	3886.0	817	8613	2218.9	86%	88%	88%
		ART	1730	1903.3	570	5057	1205.3	5108	5847.3	1669	16216	3844.5	84%	88%	88%
		CF	254	314.1	133	680	156.7	4605	5140.3	3150	8771	1623.3	84%	88%	86%
2	58%	Rand	9	9.3	4	16	2.9	35	34.8	19	53	8.6	87%	88%	88%
		ART	11	11.1	5	22	4.6	32	32.8	15	64	13.5	86%	88%	88%
		DART	428	481.3	100	1578	297.3	886	1020.4	6	4085	800.6	58%	70%	64%
		CF	8	7.8	4	10	1.5	405.4	400.5	171	606	107.5	86%	88%	87%
	86%	Rand	1428	1607.8	508	3996	736	4253	4815.5	1501	12364	2270.7	87%	88%	88%
		ART	1748	1963.1	758	5797	1031.7	5308	6019.5	2280	18325	3305.5	86%	88%	88%
		CF	877	906.7	379	2017	394.3	20600	20980.0	13477	35228	4961.3	86%	88%	87%
3	45%	Rand	16.5	17.1	9	32	5.7	48.5	52.2	27	146	23	61%	62%	61%
		ART	17.5	17.8	8	30	6.3	62	59.8	29	103	18.9	60%	62%	61%
		DART	737	693.5	1	1041	255.7	1559	1477.0	5	2571	601	45%	50%	48%
		CF	6	5.9	4	7	0.5	443	571.0	261	4247	698	55%	61%	60%
	55%	Rand	333	339.8	213	622	86.6	976	999.3	623	1832	258	61%	62%	61%
		ART	432	438.4	205	1035	148.9	1312	1324.7	619	3174	458.5	60%	62%	61%
		CF	124	133.3	82	206	29.8	10143	12314	8724	76003	12060.5	55%	61%	60%
4	50%	Rand	2	2.2	2	3	0.43	11	11.3	7	18	3.2	75%	77%	76%
		ART	2	2.3	2	3	0.46	6	7.2	6	10	1.6	75%	77%	76%
		DART	224	222.5	106	579	111.5	332	307.2	6	1362	333.7	50%	53%	52%
		CF	2	2.4	2	4	0.56	43.5	57.1	39	182	28.9	76%	78%	77%
	75%	Rand	1498	1636.7	637	3470	704.9	4518	5057.9	2188	11154	2257.1	75%	77%	76%
		ART	2230	2629.8	802	5215	2629.8	7072	8511.6	2469	17730	4564	75%	77%	76%
		CF	259	266.0	189	356	46.5	10751	10565.9	6726	13797	1941.4	76%	78%	77%
5	46%	Rand	1	1	1	1	0	11	11.7	8	27	3.5	80%	81%	80%
		ART	1	1	1	1	0	3	3.7	3	4	0.2	79%	80%	80%
		DART	1	1	1	1	0	3	3.4	3	4	0.5	46%	51%	48%
		CF	1	1	1	1	0	7	7.7	5	17	2.7	78%	79%	79%
	78%	Rand	986	1023.2	610	1558	261.8	3021	3162.5	1867	4913	842.4	80%	81%	80%
		ART	1556	1615.6	855	2717	492.4	4915	5157.7	2656	8916	1646.8	79%	80%	80%
		CF	459	463.9	312	557	54.6	19926	20040.9	13320	24408	2481.8	78%	79%	79%
6	76%	Rand	860	867.0	699	1118	113.7	2709	2721.0	2182	3562	363	78%	79%	79%
		ART	1209	1220.0	945	1522	152.5	3945	3940.0	3001	5038	524	78%	79%	78%
		CF	405	405.7	359	452	19.1	23097	22793.0	17936	25691	1754	76%	77%	77%
7	79%	Rand	526	543.1	457	714	60.6	1678	1736.8	1472	2329	207.3	82%	83%	83%
		ART	671	684.1	521	914	86.3	2167	2217.6	1675	3028	299.4	82%	83%	82%
		CF	370	380.0	311	464	41.7	18187	18829	15567	22914	1991	79%	81%	80%
8	61%	Rand	101	99.6	86	111	8	326.5	330.4	282	477	37.7	61%	61%	61%
		ART	105	107.1	90	122	8.7	353.5	358.1	301	411	28.7	61%	61%	61%
		CF	100	100.2	95	109	3.3	7804	7760.7	6982	8649	385.5	61%	61%	61%
9	64%	Rand	325	327.3	297	367	19.4	1158	1162.0	1039	1307	75.1	71%	71%	71%
		ART	406	398.9	362	440	21.6	1486	1462.0	1308	1820	107.3	71%	72%	71%
		CF	206	210.8	197	240	11.9	43685	42322.0	32908	51949	4860.6	64%	65%	65%
10	61%	Rand	372	375.4	338	423	19.1	1605	1633.6	1414	1884	112.6	65%	65%	65%
		ART	466	464.7	402	575	33.2	2099	2112.1	1781	2694	190.6	64%	65%	64%
		CF	241	241.9	222	258	9.5	66866	66837.7	55228	79976	6870.7	61%	61%	61%
11	54%	Rand	32	32.2	28	38	2.8	195	197.7	171	233	16.7	74%	74%	74%
		ART	34	33.6	29	37	2	207.5	204.3	177	277	12.5	73%	74%	74%
		CF	26	26.5	24	29	1.2	43173	41832.4	30755	47864	4399.2	54%	58%	56%
	73%	Rand	1621	1620.4	1554	1718	37.2	25530	25491.7	23465	28092	1074	74%	74%	74%
		ART	2106	2114.5	1968	2215	57.3	39045	38946.2	35005	41677	1680.5	73%	74%	74%
12	69%	Rand	1060	1067.1	954	1198	55.9	29439	29421.7	23968	36642	2549.9	70%	70%	70%
		ART	1504	1488.4	1392	1594	56.5	48050	48896.2	42367	58034	4365.1	69%	69%	69%

Table 2: Results of experiments on subject applications under test (AUTs A1–A12) with approaches (*App*): CarFast (CF), Random (Rand), ARTOO (ART) and DART. Columns C_{min}, C_{max}, and C_{med} give the minimum, maximum and median values of statement coverage after running the AUTs for 24 hours. We define the minimum value of C_{min} as the target coverage (*Cov*). We then determine how long each approach takes to reach this target. Execution times are measured in the number of iterations (*I*) and elapsed time (*E*, in seconds). For each, we report the Median, Mean, Min, Max. and the standard deviation (*SD*). AUTs A1–A5 report measurements with and without including the DART approach, but AUTs A6–A12 do not include the DART approach. For details please see Sections 4.2 and 5.2.

5.3 Our Interpretation of Results

We can summarize and interpret the results of our experiments as follows.

1. We strongly suggest that CarFast has high potential in achieving higher statement coverage faster and becoming practical especially if its execution overhead per iteration can be further reduced. We expect to reduce the overhead in the future since we found the bottlenecks of CarFast from our experiments.

2. When it comes to comparing the random approach with ARTOO, the random approach is still better when higher statement coverage is targeted. ARTOO performs as good as the random approach only when lower statement coverage is targeted. We suggest that it is likely that results depend on certain characteristics of the AUTs, finding which is a subject of future work.

6. RELATED WORK

Our approach is a test case prioritization technique: choosing an ordering of some existing test suite in order to increase the likelihood of revealing faults earlier. Elbaum et. al. [21] surveyed several approaches that effectively prioritize test cases in regression testing. These techniques use greedy algorithm to compute an order that achieves higher coverage sooner or higher fault detection rate. Thus, test coverage or fault detection rate of each test case must be known. In the context of regression testing, each test case's coverage or fault detection rate on previous versions of the AUT is used to predict their future performance. Our approach does not require prior knowledge of test cases and thus it is not restricted in the context of regression testing.

Dynamic symbolic (or "concolic") execution engines such as DART [26] generate test inputs that explore different execution paths of a program. In early work, the exploration strategy is usually depth-first or breath-first, which is the basis for many test case generation techniques, including ours.

Majumdar et. al. [43] interleave random testing and concolic exploration to improve test coverage. Their approach starts with random testing, changes to concolic exploration when random testing fails to increase coverage, and changes back to random as soon as *some* coverage is gained. In contrast, CarFast uses a systematic approach based on a static program analysis-based branch coverage gain predictor, and we evaluated CarFast with statistical significance on a large number of subject applications.

Concolic tools use different search strategies to decide how to pick branches where constrains are negated. Xie et. al. [61] use a branch's distance to the target path as the fitness function in their work. Here distance means the number of conditional control flow transfers between a branch and the target path. Branches near the target path are more likely to be picked for negation. Similarly, Burnim and Sen pick a branch when its distance to some uncovered path is small [9]. To increase coverage, Sage [27], by Godefroid et al., tries to negate not one, but as many constraints in a path condition as possible.

Our approach differs from search-based approaches in the fitness function—to the best of our knowledge, CarFast is the first tool that defines and uses a static program analysis based coverage gain predictor to guide path exploration. However, genetic-based approaches are not shown to be scalable, and they usually work on generating test data for expressions with less than 100 boolean variables. Ours is the first approach that works for large-scale applications, it is scalable, and it does not require any machine-learning algorithms, which are usually computationally intensive. In the future, once scalable genetic algorithms are developed for generating test input data for achieving higher coverage faster, we will compare CarFast with these algorithms.

7. CONCLUSION AND FUTURE WORK

We created a novel fully automatic approach for *aChieving higher stAtement coveRage FASTer (CarFast)*, by combining random testing with static program analysis, concolic execution, and constraint-based input data selection. We implemented CarFast and applied it to twelve Java applications whose sizes range from 300 LOC to one million LOC. We compared CarFast, pure random, adaptive random, and Directed Automated Random Testing (DART) against one another. The results show with strong statistical significance that when execution time is measured in terms of the number of runs of the application on different input test data, CarFast largely outperforms the evaluated competitive approaches with most subject applications.

Our experimental results are promising, and there are several areas that will improve our work. First, we plan to adapt CarFast to other test coverage metrics, such as branch coverage and basic block coverage, to study if CarFast can generalize to other metrics. Next, we plan to investigate the relationship between high coverage and fault-detection abilities with CarFast. Since there is a body of research that shows a strong correlation [51, 49, 38, 10], we expect that using CarFast increases the probability of finding faults. Finally, we plan to improve the implementation of CarFast to reduce the total elapsed time. We identified several bottlenecks from our experiments, i.e., in Dsc's current dumper mode. With more engineering on the bottlenecks, we expect CarFast to run faster and outperform random techniques with respect to both iteration and elapsed time.

8. ACKNOWLEDGMENTS

We warmly thank the anonymous reviewers for their comments and suggestions that helped us in improving the quality of this paper. This material is based upon work supported by the National Science Foundation under Grants No. 0916139, 1017633, 1217928, 1017305, and 1117369. We also acknowledge the support of Accenture, since it financed the cloud computing experiment and provided the internship opportunity for the student coauthors to work on this project.

9. REFERENCES

[1] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *ISSTA*, pages 265–275, 2011.

[2] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.

[3] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *ISSTA '10*, pages 219–230, 2010.

[4] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22:229–245, September 1983.

[5] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. 21st OOPSLA*, pages 169–190, Oct. 2006.

[6] S. M. Blackburn et al. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, Aug. 2008.

[7] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 206–212. ACM, 2005.

[8] É. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *ACM SIGOPS France (Adaptable and extensible component systems)*, Nov. 2002.

[9] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE '08*, pages 443–446, 2008.

[10] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing. In *A-MOST*, pages 1–7, 2005.

[11] M.-H. Chen, M. R. Lyu, and W. E. Wong. An empirical study of the correlation between code coverage and reliability estimation. In *3rd IEEE Int. Soft. Metrics Sym.*, pages 133–141, 1996.

[12] T. Y. Chen, R. Merkel, G. Eddy, and P. K. Wong. Adaptive random testing through dynamic partitioning. In *QSIC '04*, pages 79–86, 2004.

[13] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: Adaptive random testing for object-oriented software. In *ICSE '08*, pages 71–80, 2008.

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

[15] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic program transformation with JOIE. In *USENIX Annual Technical Symposium*, June 1998.

[16] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *ICSE*, pages 38–48, 2003.

[17] S. Cohen and B. Kimelfeld. Querying parse trees of stochastic context-free grammars. In *Proc. 13th ICDT*, pages 62–75, Mar. 2010.

[18] S. Cornett. Minimum acceptable code coverage. Bullseye Testing Technology, http://www.bullseye.com/minimum.html, 2011.

[19] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[20] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proc. 18th OOPSLA*, pages 149–168, Oct. 2003.

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

[22] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, 1981.

[23] S. Fomel and J. F. Claerbout. Guest editors' introduction: Reproducible research. *Computing in Science and Engineering*, 11(1):5–7, Jan. 2009.

[24] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *USENIX Windows Systems Symposium - Volume 4*, 2000.

[25] P. Godefroid. Compositional dynamic test generation. In *POPL '07*, pages 47–54, 2007.

[26] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI '05*, pages 213–223, 2005.

[27] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[28] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: A survey. *Softw. Test., Verif. Reliab.*, 15(3):167–199, 2005.

[29] D. Hamlet. When only random testing will do. In *RT '06*, pages 1–9, 2006.

[30] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978, 1994.

[31] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *ICS '03*, pages 63–73, 2003.

[32] I. Hussain, C. Csallner, M. Grechanik, C. Fu, Q. Xie, S. Park, K. Taneja, and B. M. M. Hossain. Evaluating program analysis and testing tools with the RUGRAT random benchmark application generator. In *WODA*, July 2012.

[33] M. Islam and C. Csallner. Dsc+mock: A test case + mock class generator in support of coding against interfaces. In *WODA*, pages 26–31, July 2010.

[34] A. Joshi, L. Eeckhout, R. H. Bell, Jr., and L. K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM TACO*, 5(2):10:1–10:33, Sept. 2008.

[35] C. Kaner. Software negligence & testing coverage. In *STAR '96*, 1996.

[36] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing*. Wiley, Dec. 2001.

[37] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley, July 2008.

[38] Y. W. Kim. Efficient use of code coverage in large-scale software development. In *CASCON '03*, pages 145–155, 2003.

[39] K. Koster and D. C. Kao. State coverage: A structural test adequacy criterion for behavior checking. In *Proc. 15th FSE, Companion Papers*, pages 541–544. ACM, Sept. 2007.

[40] R. Kuhn, Y. Lei, and R. Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10(3):19–23, 2008.

[41] C. Li and C. Csallner. Dynamic symbolic database application testing. In *Proc. 3rd DBTest*, June 2010.

[42] M. Maierhofer and M. A. Ertl. Local stack allocation. In *Proc. 7th International Conference on Compiler Construction (CC)*, pages 189–203. Springer, Apr. 1998.

[43] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE '07*, pages 416–426, 2007.

[44] Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich. Software reliability growth with test coverage. *IEEE Trans. on Reliability*, 51:420–426, 2002.

[45] B. Marick. How to misuse code coverage. In *Proc. of the 16th Intl. Conf. on Testing Comp. Soft.*, pages 16–18, 1999.

[46] G. McDaniel. *IBM Dictionary of Computing*. Dec. 1994.

[47] B. Meyer. Testing insights. *Bertrand Meyer's technology blog*, http://bertrandmeyer.com/2011/07/11/testing-insights, 2011.

[48] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[49] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *ISSTA '09*, pages 57–68, 2009.

[50] A. S. Namin and S. Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *ISSTA*, pages 342–352, 2011.

[51] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *ICSE*, pages 287–301, May 1993.

[52] P. Runeson. A survey of unit testing practices. *IEEE Softw.*, 23:22–29, July 2006.

[53] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, Nov. 1996.

[54] Sable Reserch Group. Soot: A java optimization framework. `http://www.sable.mcgill.ca/soot/`.

[55] M. Schwab, M. Karrenbach, and J. Claerbout. Making scientific computations reproducible. *Computing in Science and Engineering*, 2(6):61–67, Nov. 2000.

[56] R. M. Sirkin. *Statistics for the Social Sciences*. Sage Publications, third edition, Aug. 2005.

[57] D. R. Slutz. Massive stochastic testing of SQL. In *VLDB '98*, pages 618–622, 1998.

[58] R. Torkar and S. Mankefors. A survey on testing and reuse. In *Proc. IEEE International Conference on Software - Science, Technology & Engineering*. IEEE, 2003.

[59] Y. L. Traon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. In *ISSRE*, pages 93–102, 2007.

[60] S. Ur and A. Ziv. Off-the-shelf vs. custom made coverage models, which is the one for you? In *STAR '98*, May 1998.

[61] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, pages 359–368. IEEE, June 2009.

[62] Q. Yang, J. J. Li, and D. Weiss. A survey of coverage based testing tools. In *Proc. International Workshop on Automation of Software Test (AST)*, pages 99–103. ACM, 2006.

[63] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.