Achieving Higher Test Coverage Faster

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

Ishtiaque Hussain, Christoph Csallner
University of Texas at Arlington
Arlington, TX 76019-9800
ishtiaque.hussain@mavs.uta.edu,csallner@uta.edu

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

Kunal Taneja
North Carolina State University
Raleigh, NC, USA
ktaneja@ncsu.edu

Mark Grechanik
Accenture Technology Labs and
U. of Illinois, Chicago, IL 60601
drmark@uic.edu

Chen Fu, Qing Xie
Accenture Technology Labs
Chicago, IL 60601
{chen.fu,qing.xie}@accenture.com

Abstract—Test coverage is an important metric of software quality, since it indicates thoroughness of testing. A fundamental problem of software testing is how to achieve higher coverage faster, and it is a difficult problem since it requires testers to cleverly pinpoint test input data to steer execution sooner toward application code that contains more statements.

We created a novel fully automatic approach for ensuring that test Coverage Achieved higher and FASTER (CarFast), which we implemented and evaluated on twelve Java applications whose sizes range from 300 LOC to one million LOC. We compared CarFast and pure random and adaptive testing 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 evaluated competitive approaches with most subject applications.

I. INTRODUCTION

Test coverage is an important metric of software quality [58], since it indicates thoroughness of testing. Specifically, achieving higher test coverage is correlated with the probability of detecting more defects [11], [37], [46], [48] and increasing reliability of software [12], [41]. Even though it is agreed that test coverage alone may not always be a strong indicator of software quality [35, page 181], it is a general consensus that achieving higher test coverage is desirable for gaining confidence in software [12], [48]. Achieving higher test coverage means that testers select test input data with which they can execute larger portions of application code. Higher coverage is always better for increasing the confidence of stakeholders in the quality of software; however, 100% coverage is rarely achieved especially when testing large-scale applications [42], [48].

The faster these testers achieve higher coverage, the lower is the cost of testing [34], 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 test 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. In this paper we concentrate on statement coverage, which measures the percentage of the executed statements to the total number of statements in the AUT [58]. We chose statement coverage since it is simpler to reason about and it is viewed, among other things, as effective in isolating portions of code that could not be executed [5, page 90].

A big and important challenge is to get higher coverage faster for nontrivial applications with 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 values of the 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 input parameters. Having an insight into how to choose a small subset of these combinations that results in the maximum coverage enables achieving the highest coverage faster. Unfortunately, dealing with an enormous number of combinations of values is infeasible even for a small number of inputs; for example, 20 integer inputs whose values range from zero to nine give us a combination space of the size 10^{20}. Knowing what test input data to select to drive the AUT towards different branches is very difficult. Thus, a fundamental problem of software testing is how to achieve higher coverage faster by selecting specific values of input parameters that lead the executions of applications to cover more statements in a shorter time period.

We created a novel approach for ensuring that test Coverage Achieved higher and FASTER (CarFast) using the intuition that higher 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 branch dependent* [45]. In CarFast, static analysis is used to approximately estimate the number of statements that are contained within AUT branches. Once it is known which branches contain more statements, CarFast uses a constraint-based selection approach [39] to select input data that will guide the AUT towards these branches. Unlike CarFast, different variations of automatic test data generation techniques use computationally expensive constraint solvers [15], [19], [25], [40], and these solvers negatively affect the scalability of these techniques. In replacing constraint solvers with selectors, not only does CarFast achieve higher coverage faster, but also it achieves better scalability as it is shown in Section V.

CarFast offers multiple benefits: it is fully automatic since it does not require any intervention by testers, it is tractable since it requires exploration of 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 as demonstrated by our experiments on large-scale applications with up to 500KLOC. Even though we build CarFast to work with Java programs, there are no fundamental limitations to generalize it to other languages and platforms.

This paper makes the following contributions:

- We developed a novel algorithm for achieving higher test coverage faster and we implemented it as part of CarFast. CarFast is the first approach that predicts test coverage gain using static program analysis to guide path exploration. Unlike other concolic engine-based approaches that use constraint solvers, we designed and implemented a *constraint-based selection* that queries a database to find input data values with significantly less overhead, making our solution more scalable.
- We implemented CarFast and applied it to twelve Java application benchmarks whose sizes range from 300 LOC to one million LOC that we generated using stochastic parse trees [55]. CarFast, subject Java applications, and the stochastic Java application benchmark generator are available for public use.
- We conducted a large-scale experiment using Amazon EC2 cloud to evaluate CarFast and competitive approaches against one another, specifically, pure random and adaptive random testing, and Directed Automated Random Testing (DART) [26]. The results show with strong statistical significance that with CarFast higher coverage is achieved faster for smaller applications while the random testing approach outperforms all other approaches for large applications when execution time is measured in terms of elapsed time rather than the numbers of iterations of executions of AUTs with different input values. However, when execution time is measured in terms of iterations, CarFast outperforms all evaluated competitive approaches with all subject AUTs with strong statistical significance.

---

```java
if ( i1 == 10 ) {
    // branch 1: 300 statements
    . . .
} else if ( i2 == 50 ) {
    // branch 2: 600 statements
    . . .
} else {
    // branch 3: 100 statements
    if( . . . ) { if( . . . ){
        . . . } . . . }
}
```

Fig. 1: An illustrative example.

- Finally, for smaller applications, adaptive random performs statistically as good as pure random testing, while for large applications pure random testing beats adaptive random with strong statistical significance. In addition, DART performs the worst for all AUTs in our experiments.

II. OUR 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.

A. An Illustrative Example

An illustrative example is shown in Figure 1 using Java-like pseudocode. Line numbers to the right are used simply for the convenience of references. This example has *if-else* statements that control three branches, where branch numbers are shown in comments along with the numbers of statements that these branches encapsulate. 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 7–10. The number of distinct uncovered statements that are reachable from this branch is 100, which is significantly less than the numbers of statements that two other branches encapsulate. We say that a previously uncovered statement is reachable from a branch *B* if there exists a concrete input that triggers an execution that evaluates the condition of the branch *B* and covers *S*.

In addition, this branch contains complex control-flow, such as several nested *if-else* statements that contains arbitrary numbers of statements that add up to 100. 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 \( i1 \neq 10 \land i2 = 50 \) to steer the next run toward branch 2 in lines 4–6, since it contains the biggest number of distinct reachable uncovered statements (i.e., 600 statements as it is shown in line 5 in Figure 1), 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

---

1http://www.carfast.org
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 the branch 3 using a depth-first search algorithm. That is, instead of finishing the first run through branch 3 quickly and moving on to more profitable branches in terms of increased coverage, DART keeps exploring the branch for a while even though the gain in coverage will rather be minuscule.

B. Our Hypothesis and Approach

We hypothesize that we can significantly improve the speed with which higher coverage is achieved by guiding the selection of test input data using estimations of the numbers of distinct statements that are still uncovered in different branches and constraints on inputs that are collected as the artifacts of executing the AUT. This hypothesis is rooted in the essence of systematic testing, which is a counterpart to the most, we rank these branches by counting the statements. For example, if the first input data is selected executed branches or branches that have smaller numbers of statements in the AUT, and if constraints from branch conditions enable selection of test input data that lead to execute statements within those branches.

We review how our approach works by considering a code fragment that is shown in 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., 600 statements as it is shown in line 5 in Figure 1), constraint $C_2$ can be negated and the resulting constraint formula will be $C_1 \land \neg C_2$ or $i1 \neq 10 \land i2 = 50$. 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.

Of course, in the worst case random selection may result in test input data that will lead execution towards already executed branches or branches that have smaller numbers of 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 [4], [29], [30]. 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.

C. Ranking Branches

To understand which branches affect the coverage of the AUT the most, we rank these branches by counting the numbers of statements that these branches contain. Executing higher ranked branches enable achieving higher coverage. To obtain ranks for branches, we construct and traverse a control-flow graph (CFG) of the AUT, and we count the number of statements that are control-dependent on branch conditions and reachable from the top-level branch condition. For method call statements, statements are counted inside target methods as well. Specifically, we perform virtual call resolution using static class hierarchy analysis, and we count all the statements in all the target methods, but only when the call site is the only entry point of that method. Currently, we only take into consideration that values of input parameters are used in variables that control branches. Extending it to compute how many statements are affected by these input parameters precisely is a subject of future work.

D. Sources of Test Input Data

The input test data comes from existing repositories or databases; it 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, Renters Insurance Program designed and built by a major insurance company has a database that contains approximately 78Mil customer profiles, which are used as the test input data for different applications including Renters Insurance Program. As part of CarFast, we translate constraints into SQL queries against these databases [39]. For example, to find values for input variables $i1$ and $i2$ that satisfy the constraint $i1 \neq 10 \land i2 = 50$, the following SQL query is executed: SELECT * FROM InputTbl WHERE $i1 \neq 10 \land i2 = 50$. Of course, these SQL queries include thousands of conditions in the WHERE clause, and to scale CarFast we develop a lightweight and efficient SQL-based constraint evaluator that we describe in Section III-C.

E. 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 coverge $totalCov$ 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 computed 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.
Algorithm 1 The CarFast algorithm.

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

This algorithm runs the loop between steps 7–30, which terminates on the condition of the reached time limit or if desired coverage (i.e., the predefined value \text{covCeiling}) is achieved. In the step 8 the AUT, } P \text{ is executed using the input } t \text{ resulting in the updated value of the } \text{totalCov}, \text{ added branches that were covered during this execution to the set of covered branches, } B_{\text{cov}}, \text{ and added constraints that have been learned during this execution. Then, in the } \text{for} \text{ loop in steps} \\
10–24, \text{ each member branch in the set } B_{s} \text{ is examined to check to see if it was covered in the previous run of the AUT.} \\
\text{If some branch, } b_k \text{ was covered, it is removed from the set } B_{s} \text{ in line 23, otherwise, one or more of the constraints, } C_k \text{ \text{are negated or flipped in line 12. By treating a constraint as a} } \\
\text{query to obtain input data that satisfy the conditional WHERE} \\
\text{clause, the subsets of test input data, } \{t_c\} \text{ is obtained in line} \\
13 \text{ that satisfy this clause, that is the flipped constraint. If} \\
\{t_c\} \text{ is empty, then no test input data from our input database} \\
\text{can lead to the desired branch, and this message is issued in} \\
\text{line } 20. \text{ Otherwise, one input, } t \text{ is randomly selected from the} \\
\text{set } \{t_c\} \text{ in line 15 and the control is returned to line 26 and} \\
\text{eventually to line 8, where the AUT is run with this input} \\
\text{thereby repeating the loop.} \\
\text{In some cases, it may not be possible to know exact} \\
\text{constraints to reach certain statements, since the set of con-} \\
\text{straints that is collected by the concolic engine corresponds} \\
\text{to reaching different nodes of the CFG, and subsequently} \\
\text{different statements. Flipping these constraints and solving} \\
\text{these flipped constraints may result in input data that will lead} \\
\text{the AUT toward other uncovered statements, but not necessarily} \\
\text{the desired statements. However, as more constraints are} \\
\text{collected with newly obtained test input data, these constraints} \\
\text{will eventually enable CarFast to narrow down the scope of} \\
\text{the executed statements to the ones that are desirable. Of} \\
\text{course, this works only if the Input Test Data set contains} \\
\text{an input that can reach such statements. If the application} \\
\text{contains a statement that is not reachable with the inputs from} \\
\text{Input Test Data, then CarFast will never cover that statement.} \\
\text{However, as the results of the experimental evaluation show in} \\
\text{Section V, CarFast outperforms other competitive approaches} \\
\text{under different conditions. In general, for a large mature} \\
\text{application in its } N^{\text{th}} \text{ release, our experience is that the input} \\
\text{data base covers the most important statements, as generations} \\
\text{of maintainers and testers have added test cases that cover both} \\
\text{the important business cases as well as tricky corner cases that} \\
\text{have caused problems in the past.} \\

III. IMPLEMENTATION

In this section we describe the salient features of our implementation including the concolic engine, the constraint- 

based selection.

A. CarFast Implementation and Deployment Challenges

Our implementation goal is to demonstrate that CarFast is viable by applying it to large-scale AUTs. CarFast poses 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.

B. Dynamic Symbolic (Concolic) Execution Engine

We used DSC [32], a Java dynamic symbolic execution 

engine (i.e. concolic engine) for Java AUTs in CarFast. Below we describe its main features and how we adapted it to scale to large AUTs.
1) Overview of DSC: DSC instruments the bytecode of AUTs automatically, that is, it inserts method calls (i.e., callbacks) after each instruction in the code. During AUT execution, the callbacks enable the DSC to maintain the symbolic state by mirroring the effect 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 existing Java execution environments; it does not require any modifications of 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 [16], using the third-party open source bytecode instrumentation framework ASM [9]. 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.

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 available memory.

To scale to large applications, we introduce a Dumper mode for DSC to minimize the memory consumption for the symbolic state representation. Instead of caching symbolic expressions to 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.

Consider the expression \( a = (a-5) + (b/3) \) and the following statement \( \text{if} (a > 5) \text{...} \). DSC represents input parameters \( a \) and \( b \) as two symbolic variables (e.g., \( \text{var0} \) and \( \text{var1} \)) and monitors the execution. In the normal mode, for the assignment statement in the example, DSC creates a corresponding symbolic expression \( \text{var0} - 5 \) + \( \text{var1}/3 \) and cache it in the memory. Assuming the branch statement evaluates to \( \text{true} \), it would create a constraint using cached expression for \( a \) of the form: \(((\text{var0} - 5) + (\text{var1}/3)) > 5\). To minimize the memory consumption for the symbolic state representation and to scale for large programs, DSC in the dumper mode replaces each expression with a symbol and writes this expression onto a disk.

Later, to create the constraints involving initial symbolic variables for the input parameters, DSC performs a single pass over the saved expressions. In this pass, each assignment statement (e.g., \( v1 = (\text{var0} - 5) \)) is stored in a memory table, and for each condition statement (e.g., \( v3 > 5 \)), dynamic table lookup finds replacement for variables to recover the constraints dynamically thereby achieving better scalability.

C. 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, the concolic engine is 32-bit meaning that it can use less than four gigabytes of RAM. 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 and deployed it in a cloud computing environment. As part of the future work, we will use multiple constraint selector servers that can be started on demand, thereby improving response time and scalability of CarFast.

D. Miscellaneous

We implement branches ranking using Java static analysis and transformation engine Soot [51]. All conditional branch statements in AUTs are ranked using the approach that we described in Section II-C. At runtime, we uses EMMA [49] to compute and report statement coverages. Also, we modified callback functions in DSC to keep track of covered branches during test execution to reset rankings of the already executed branches to zero to avoid repeatedly executing already covered statements.

IV. Experiments

To determine how effective CarFast is in achieving higher test 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 explain these competitive approaches, describe the methodology of our experimental design, explain our choice of subject AUTs, and discuss threats to validity.

A. Variables

The main independent variable is 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). Dependent variables are the execution time that it takes to achieve this 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, \( I \). While the elapsed time gives the absolute value of the time it takes to reach a certain level of coverage, measuring the number of iterations provides an insight into the potential of a given approach. For example, even if it takes longer to achieve some level of coverage, however with fewer iterations, it may
be possible to further reduce the time it takes to execute AUT per iteration thereby evolving an approach to be more efficient in future. Thus, if an approach achieves a higher test coverage using fewer iterations, however longer elapsed time it takes per iteration, this time can eventually be reduced by improving efficiency of the particular approach. 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.

1) Random Testing: Random testing approach, as the name suggests, involves random selection of test input data for input parameter values, and that it showed remarkably effective and efficient for exploratory testing and bug finding [6], [24]. A seemingly “stupid” idea of random testing proved often more effective than systematic sophisticated testing approaches [29], [30]. To prove our claims in this paper, our goal is to show under what conditions CarFast outperforms random testing with strong statistical significance.

2) Adaptive Random Testing: Adaptive random testing (ART) is a controversial refinement of the baseline random testing where randomly selected data are distributed evenly across the input data space [13]. 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 [14]. 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, Briand et al presented statistically significant results of experiments that question the effectiveness of ARTOO with respect to bug detection [2], and Meyer pointed out in his response that ARTOO should be evaluated further [44]. 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.

3) DART: Directed Automated Random Testing (DART) is an approach that uses a concisic engine to generate test inputs that explore different execution paths of a program [26]. In the original DART algorithm, the 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.

B. Methodology

Our goal is to determine with which approach higher test coverage can be achieved faster. Given the complexity of 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 test 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.

In designing the methodology for this experiment we aligned with the guidelines for statistical tests to assess randomized algorithms in software engineering [3]. 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 run 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 11 AUTs we run 30 experiments for each of four approaches, results in a total of $11 \times 4 \times 30 = 1,320$ experiment runs.

To evaluate our hypotheses we run statistical tests are based on an 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 [54, 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.

Experiment are carried out in large instances of a virtual machine at Amazon EC2 with the following configuration: 7.5 GB RAM, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB instance storage. For each of 30 runs of each experiment with each AUT, we run it for 24 hours of time limit (which is chosen experimentally) to establish what coverage can be achieved for this AUT. That is, the total execution time is $1,320 \times 24 = 31,680$ hours. With the cost of $0.48 per instance per hour as of September, 2011, the total cost of this experiment is over USD $15,200 not to mention the additional cost for disk space, I/O requests, and the internet traffic. We determine the maximum coverage for each run within each experiment that was reached after running for 24 hours, and then we select the minimum of the maximum coverages reached for a set of 30 runs for an experiment. This way we experimentally establish the ceiling for test coverage so that we can retrieve times it takes to reach this coverage in each run.

C. 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 control group, and other competitive approaches are applied to AUTs in the control

2http://aws.amazon.com/ec2/instance-types as of Sept 20, 2011
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, \( \mu \), 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:

\( H_1 \): CarFast versus Random. The effective null hypothesis is that \( \mu_{S_{\text{CarFast}}} = \mu_{S_{\text{Rand}}} \), while the true null hypothesis is that \( \mu_{S_{\text{CarFast}}} \leq \mu_{S_{\text{Rand}}} \). Conversely, the alternative hypothesis is \( \mu_{S_{\text{CarFast}}} > \mu_{S_{\text{Rand}}} \).

\( H_2 \): CarFast versus ARTOO. The effective null hypothesis is that \( \mu_{S_{\text{CarFast}}} = \mu_{S_{\text{AR TOO}}} \), while the true null hypothesis is that \( \mu_{S_{\text{CarFast}}} \leq \mu_{S_{\text{AR TOO}}} \). Conversely, the alternative hypothesis is \( \mu_{S_{\text{CarFast}}} > \mu_{S_{\text{AR TOO}}} \).

\( H_3 \): CarFast versus DART. The effective null hypothesis is that \( \mu_{S_{\text{CarFast}}} = \mu_{S_{\text{DART}}} \), while the true null hypothesis is that \( \mu_{S_{\text{CarFast}}} \leq \mu_{S_{\text{DART}}} \). Conversely, the alternative hypothesis is \( \mu_{S_{\text{CarFast}}} > \mu_{S_{\text{DART}}} \).

\( H_4 \): ARTOO versus Random. The effective null hypothesis is that \( \mu_{S_{\text{AR TOO}}} = \mu_{S_{\text{Rand}}} \), while the true null hypothesis is that \( \mu_{S_{\text{AR TOO}}} \geq \mu_{S_{\text{Rand}}} \). Conversely, the alternative hypothesis is \( \mu_{S_{\text{AR TOO}}} < \mu_{S_{\text{Rand}}} \).

The rationale behind the alternative hypotheses to \( H_1, H_2, \) and \( H_3 \) is that CarFast achieves certain test coverage faster than other approaches. The rationale behind the alternative hypothesis to \( H_4 \) is that the random approach outperforms ARTOO as suggested by Briand et al [2].

D. Subject Applications

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 [43]. Benchmarks are very important for evaluating program analysis and testing approaches based on the variability of results for applications that have different properties [3].

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 [56]. 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] to demonstrate the effectiveness...
of the proposed approach or as the benchmark to test the performance of some system [31].

2) Random Benchmark Applications: A random program can be defined by construction. Consider that every program is an instance of the grammar of the language in which this program is written. Typically, grammars are used in compiler construction to write parsers that check the syntactic validity of a program and transform its source code into a parse tree [1]. An opposite use of the grammar is 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. When more than one production rule can be chosen to replace a nonterminal, a rule is chosen based on the probability that is assigned to a rule, the more frequent its metrics among others. These grammars and parse trees are also in generating SQL statements for testing database engines [56]. With the stochastic grammar model it is ensured that the generated program is syntactically correct.

3) Subject AUTs For Experimentation: We generated twelve subject AUTs whose sizes range from 303 LOC to over one million LOC. Table I 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.

<table>
<thead>
<tr>
<th>Subject AUT</th>
<th>Size [kLOC]</th>
<th>NOC</th>
<th>NOM</th>
<th>NBD</th>
<th>MCC</th>
<th>WMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA1</td>
<td>0.303</td>
<td>4</td>
<td>3</td>
<td>2.47/6</td>
<td>6.3/20</td>
<td>23.5/36</td>
</tr>
<tr>
<td>BA2</td>
<td>0.603</td>
<td>5</td>
<td>5</td>
<td>2.3/5</td>
<td>10.4/30</td>
<td>33.4/56</td>
</tr>
<tr>
<td>BA3</td>
<td>1.23</td>
<td>14</td>
<td>19</td>
<td>2.04/5</td>
<td>6.9/26</td>
<td>24.2/44</td>
</tr>
<tr>
<td>BA4</td>
<td>1.3</td>
<td>18</td>
<td>61</td>
<td>2.16/9</td>
<td>3.78/14</td>
<td>22.4/47</td>
</tr>
<tr>
<td>BA5</td>
<td>2.13</td>
<td>24</td>
<td>49</td>
<td>2.02/5</td>
<td>4.5/13</td>
<td>24.5/36</td>
</tr>
<tr>
<td>BA6</td>
<td>5.19</td>
<td>37</td>
<td>184</td>
<td>2.02/8</td>
<td>5.2/23</td>
<td>42.9/86</td>
</tr>
<tr>
<td>BA7</td>
<td>7.81</td>
<td>38</td>
<td>469</td>
<td>2.21/8</td>
<td>4.3/19</td>
<td>63.4/137</td>
</tr>
<tr>
<td>BA8</td>
<td>24.17</td>
<td>111</td>
<td>765</td>
<td>2.4/8</td>
<td>4.7/23</td>
<td>66.4/102</td>
</tr>
<tr>
<td>BA9</td>
<td>46.65</td>
<td>61</td>
<td>428</td>
<td>4.2/12</td>
<td>22.3/56</td>
<td>249.2/347</td>
</tr>
<tr>
<td>BA10</td>
<td>98.43</td>
<td>96</td>
<td>1,576</td>
<td>3.4/8</td>
<td>10.7/27</td>
<td>325.3/447</td>
</tr>
<tr>
<td>BA11</td>
<td>470.8</td>
<td>311</td>
<td>2,244</td>
<td>4/7</td>
<td>34.1/93</td>
<td>464.3/640</td>
</tr>
<tr>
<td>BA12</td>
<td>1,157.2</td>
<td>781</td>
<td>17,449</td>
<td>3.8/13</td>
<td>13/47</td>
<td>486/631</td>
</tr>
</tbody>
</table>

TABLE I: Characteristics of the subject benchmark AUTs (BA). The format avg/max shows average and maximum values. Size = AUT code size, NOC = number of classes, NOM = number of methods, NBD = nested block depth, MCC = McCabe cyclomatic Complexity, WMC = Weighted Methods per Class.

E. Input Test Data For Subject AUTs

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 select a large subset of input data objects for testing, 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 [17], [28], [38]. 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. We generate data for our experiments using pairwise testing from the range of input date $[−50, 50]$ that was chosen experimentally.

F. Threats to Validity

The main threat for our experimental design is the selection of subject AUTs and their characteristics. Due to limitations of dynamic symbolic engine, 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 test coverage rather than using some fault detection metric. In general, even though a connection exists between test 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 [47]. Finally, test coverage is also an important metric for stakeholders to obtain
confident 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.

V. RESULTS

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 coverages 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 $H_1$.

The results of t-tests of hypotheses $H_1$–$H_4$ for paired two sample for means for two-tail distribution are shown in Table II. Statistical results for execution times are shown in Table III. DART ran out of memory for AUTs BA6–BA12 and CarFast ran out of memory for the AUT BA12. We can summarize these results as following.

- When only iterations are counted, CarFast achieves higher coverage faster when compared with the random and ARTOO approaches for all AUTs but BA4 and BA5, with strong statistical significance. AUTs BA4 and BA5 do not outperform CarFast; in fact, their execution times are close to the one of CarFast. Our explanation is that it takes around one or two iterations for these AUTs to achieve the desired level of coverage in most cases, thereby making it difficult for CarFast to outperform competitive approaches. When comparing CarFast with DART using iteration count, CarFast outperforms DART for all AUTs but BA5, which is easily explained since both approaches reach the desired coverage in one iteration in most cases.

- When only elapsed execution times are counted, random and ARTOO achieve higher 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 BA5, which is easily explained since both approaches reach the desired coverage in one iteration in most cases.

- When only iterations are counted, the random approach achieves higher coverage faster when compared with the ARTOO approach for all AUTs but BA1–BA5, with strong statistical significance. We suggest that for small applications whose size is less than three KLOC, the random approach does not outperform ARTOO. As we observe from the absolute values in Table III, ARTOO median iteration values are slightly higher than that of the random approach, meaning that latter does not perform worse that ARTOO.

- When only elapsed execution times are counted, the random approach achieves higher coverage faster when compared with the ARTOO approach for all AUTs but BA1–BA5 and BA11, with strong statistical significance. This result is somewhat surprising since we found no correlation between different software metrics for the subject AUTs that we showed in Table I.

Our interpretation of results. The results of our experiments strongly suggest that CarFast has high potential in achieving higher coverage faster and becoming practical especially if its execution overhead per iteration can be further reduced. When it comes to comparing the random approach with ARTOO, the random approach still is better especially when evaluated on large applications. We suggest that it is likely that results depend on certain characteristics ofAUTs, finding which is a subject of future work.

VI. RELATED WORK

Our approach is a test case prioritization techniques: 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 is not restricted in the context of regression testing.

Dynamic symbolic execution engines (also called concolic engines) [26], [53] are used to generate test inputs that explore different execution paths of a program. In early works the exploration is usually conducted in Depth-First-Order (DFO) or Breath-First-Order (BFO). This is the basis for many test generation techniques, including ours.

Majumdar et. al. [40] use interleaving of random testing and concolic path exploration to improve test coverage. Their approach starts with random testing, changes to concolic path exploration when random testing fails to increase coverage after a certain number of iterations, and changes back to random as soon as some coverage gain is realized using concolic test case generation. Their goal is to achieve high test coverage. Our goal is to achieve higher test coverage using minimum amount of test cases.

An important difference among many concolic test case generation tools is their search strategies, or fitness functions that decide how to pick branches where constraits are negated. Xie et. al. [57] described a system that generates test case to cover a given program path. A branch's distance to the target path is used 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
SAGE, which tries to negate not one, but as many constraints as possible. Godefroid et al. [27] proposed concolic test generation tool CAR that tries to negate the constraints. We will compare CAR with other tools like Sen’s work [10], a branch is picked for negation when its distance to some uncovered path is small. This search strategy is well suited for their goal: achieving better branch coverage. However, genetic-based approaches are not 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 CAR with these algorithms.

## VII. Conclusion

We created a novel fully automatic approach for ensuring that test Coverage Achieved faster (CAR) by combining random testing with static program analysis, concolic execution, and constraint-based input data selection. We implemented CAR and applied it to twelve Java applications whose sizes range from 300 LOC to one million LOC. We compared CAR and pure random and adaptive testing 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 data, CAR largely outperforms evaluated competitive approaches with most subject applications.

### References


---

**TABLE III: Results of experiments with subject AUTs with execution time measured in the number of iterations, for different values of the maximal achieved coverage specified in the column Cov for different approaches specified in the column Approach, whose measurements are reported in the following columns: Time, Median, Means (μ), Max, and the standard deviation, SD.**
