Improving Search-based Test Suite Generation with Dynamic Symbolic Execution

Juan Pablo Galeotti
Saarland University
Saarbrücken, Germany
galeotti@cs.uni-saarland.de

Gordon Fraser
University of Sheffield
Sheffield, UK
Gordon.Fraser@sheffield.ac.uk

Andrea Arcuri
Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

***Abstract*—Search-based testing can automatically generate unit test suites for object oriented code, but may struggle to generate specific values necessary to cover difficult parts of the code. Dynamic symbolic execution (DSE) efficiently generates such specific values, but may struggle with complex datatypes, in particular those that require sequences of calls for construction. The solution to these problems lies in a hybrid approach that integrates the best of both worlds, but such an integration needs to adapt to the problem at hand to avoid that higher coverage in a few corner cases comes at the price of lower coverage in the general case. We have extended the Genetic Algorithm (GA) in the EvoSuite unit test generator to integrate DSE in an *adaptive* approach where feedback from the search determines when a problem is suitable for DSE. In experiments on a set of difficult classes our adaptive hybrid approach achieved an increase in code coverage of up to 63% (11% on average); experiments on the SF100 corpus of roughly 9,000 open source classes confirm that the improvement is of practical value, and a comparison with a DSE tool on the Roops set of benchmark classes shows that the hybrid approach improves over both its constituent techniques, GA and DSE.**

I. Introduction

Generating unit test suites automatically is an important contribution towards improving software quality, and techniques like search-based software testing [18] (SBST) and dynamic symbolic execution [9] (DSE) can efficiently produce test suites achieving high code coverage. However, there are limitations: SBST is based on heuristics and may be inefficient when the heuristics and search operators do not favour the problem at hand. DSE, on the other hand, is dependent on the capabilities of the underlying constraint solver. Furthermore, DSE does not cope well when it comes to creating complex objects through sequences of method calls [26], and generally expects a test driver that provides a single entry point. The problems of SBST and DSE are orthogonal, suggesting that a combination may produce better results in unit test generation than the individual parts.

For example, consider the `Bar` class in Figure 1: SBST will quickly create a test suite that calls all methods, and it can easily generate instances of the `Foo` dependency class. However, optimizing the input string to the constructor of `Bar` may take a long time: If evolutionary search (e.g., a Genetic Algorithm) is applied to generate test suites, then each statement in a test case only has a small probability of being mutated at every step, but to optimize a string from any given value to the value "baz" may require many mutations. In contrast, standard DSE tools may be unable to create `Foo` instances in the first place, and would not be able to create a sequence of five successive calls to its `inc` method, which is required to cover the target branch. Where the individual techniques fail, only a combined approach may succeed.

```
class Foo {
  int x = 0;

  void inc() {
    x++;
  }

  int getX() {
    return x;
  }
}
```

```
class Bar {
  String x;
  Bar(String x) {
    this.x = x;
  }

  void coverMe(Foo f) {
    String y = x+f.getX();
    if(y.equals("baz5"))
      // target
  }
}
```

Fig. 1. Search-based testing will quickly cover all methods of `Bar`, but optimizing the input string of `Bar` to the value "baz" may take a significant amount of time. Given a string constraint solver, Dynamic Symbolic Execution may generate "baz" as an input value, but will not succeed in calling `inc` five times, which is required to cover the target branch. A hybrid approach overcomes these issues.

Forays into combining SBST and DSE have led to promising initial results (e.g., [11], [15], [17], [23]), yet the success of such combinations is *highly problem specific*: Some classes may require mostly sequences of calls to achieve high coverage, while others may represent numerical problems perfectly suited for approaches using constraint solvers, i.e., DSE. A hybridization of SBST and DSE also necessitates a large number of parameters — how are the techniques integrated, when is which technique applied, how much resources are devoted to each technique? Unfortunately, a wrong choice of parameters for a problem at hand may have a detrimental effect, leading to even worse coverage than if the constituent techniques would have been applied on their own.

To overcome these problems, we present an *adaptive* approach that combines a Genetic Algorithm (GA) used in a whole test suite generation approach with DSE: At a high level, SBST is applied, but during this search-based exploration we determine whether the problem at hand is potentially suitable for DSE or not. Mutations on primitive

values performed as part of the GA give hints on whether the problem at hand is suitable for DSE, and if so we optimize these primitive values using DSE.

We have implemented our approach as an extension to the EVOSUITE unit test generation tool, and experiments on a set of difficult classes, the SF100 corpus of open source classes [6], and the Roops[1] benchmark all confirm a significant increase in coverage.

In detail, the contributions of this paper are:

DSE in whole test suite generation: We present a novel and adaptive approach to integrate dynamic symbolic execution in a search for test suites.

Evaluation: We evaluate our approach on 38 "difficult" open source classes, the large SF100 corpus of classes, and the Roops benchmark, and compare it to standard SBST implemented in EVOSUITE, as well as the DSE tool DSC [12].

II. BACKGROUND

A simple but effective technique to generate test data for code coverage is to randomly execute a program. Random testing tools such as Randoop [20] have become very popular, are easy to implement, and have very little computational overhead. However, as such approaches tend to exercise mainly the "shallow" regions of a program and struggle to reach more difficult parts, more sophisticated techniques have been devised. In the area of test data generation for code coverage, the recently most successful techniques are *dynamic symbolic execution* (e.g., DART [9], CUTE [22]) and *search-based software testing* [18].

A. Search-based Testing

Search-based testing (SBST) describes the use of efficient search algorithms for the task of generating test cases. One of the most commonly applied global search algorithms is a *Genetic Algorithm* (GA). A GA tries to imitate the natural processes of evolution: An initial population of usually randomly produced candidate solutions is evolved using search-operators that resemble natural processes. Selection of parents for reproduction is done based on their fitness (survival of the fittest). Reproduction is performed using crossover and mutation with certain probabilities, and the operators used depend on the chosen representation. With each iteration of the GA, the fitness of the population improves, until either an optimal solution has been found, or some other stopping condition has been met (e.g., maximum time or number of fitness evaluations). In evolutionary testing, the population would for example consist of test cases, and the fitness estimates how close a candidate solution is to satisfying a coverage goal.

A fitness function guides the search in choosing individuals for reproduction, gradually improving the fitness values with each generation until a solution is found. For example, to generate tests for branch coverage a common fitness function [18] integrates the *approach-level* (number of unsatisfied control dependencies) and the *branch distance* (estimation of how close the deviating condition is to evaluating as desired). Such search techniques have not only been applied in the context of primitive datatypes, but also to test object-oriented software using method sequences [8], [25].

The traditional approach to SBST is to optimize a test case for each coverage objective in isolation. This makes it problematic to distribute a limited amount of computational resources on a set of coverage objectives, in particular considering that some of these objectives may be infeasible. Whole test suite generation [7] optimizes an entire test suite at once towards satisfying a coverage criterion, instead of considering distinct test cases directed towards satisfying distinct coverage goals. This means that the result is neither adversely influenced by the order nor by the difficulty or infeasibility of individual coverage goals.

B. Dynamic Symbolic Execution

DSE uses symbolic execution to gather path conditions from concrete executions. A path condition is a logical condition on the input values, such that a model for the condition is a program input that follows the path described by the condition. DSE usually starts with a random input, and produces the path condition $P = p_1 \wedge p_2 \wedge \ldots \wedge p_n$ for this input. Each p_i represents a branching condition in the code. For example, if there is a branch on the input variable x: `if x == 5`, then the path condition at this point will be extended with either $x == 5$ or $x \neq 5$, depending on the actual evaluation of the predicate during the concrete run. By negating an individual p_i for $i \leq n$ one can produce a new path condition $P' = p_1 \wedge \ldots p_{i-1} \wedge \neg p_i$, such that an input that satisfies P' leads to execution of a different path than P. A constraint solver can be queried with this condition and derives a new test input. This is done systematically until no further branches can be negated, i.e., all paths have been explored. This approach has been popularized in particular by the recent development of powerful SMT solvers.

An issue with SBST is that every time the fitness is evaluated, we need to re-execute a test case. In particular when it comes to search on strings, many fitness evaluations are necessary, which can be problematic. This is a problem specific to SBST; in DSE, once the path condition has been collected, the search for new inputs is performed entirely by a constraint solver, and a new execution will be launched if and only if the constraint solver successfully finds a new set of input values.

C. Hybrid Approaches

As the idea to combine SBST and DSE is appealing, there have been several attempts to combine these techniques.

The first direction of integrating the two techniques is by exploiting the information offered by path conditions to improve the search. Malburg and Fraser [17] use a GA to derive test data, but apply a special mutation operator where the path constraints are collected for the individual

[1] http://code.google.com/p/roops/

that should be mutated, and then like in DSE one path condition is mutated; the resulting constraint system is solved with a constraint solver and represents the mutated test case. Baars et al. [3] introduced the idea of symbolic search-based testing, where the fitness function of the GA takes different possible symbolic paths to the target into account. Sakti et al. [21] integrate constraint solving into the search-based test generator eToc [25], such that the individuals of the search that satisfy constraints based on relaxed versions of the program code. Vice versa, fitness information can be used during DSE exploration: The Fitnex approach [27] uses a fitness value based on branch distances [18] to select the next path condition during DSE exploration.

A second direction of integrating search and DSE is in terms of using SBST techniques to solve constraints that traditional constraint solvers may struggle with. Lakhotia et al. [15] extended the DSE tool PEX to use local search to solve floating point constraints, which constraint solvers struggle with. The CORAL tool [4], [23] applies global and local search to solve complex mathematical constraints in the context of Symbolic JPF.

The third type of integration aims to tightly integrate the two approaches, such that the resulting approach can switch between DSE and search. Inkumsah and Xie [11] proposed a combination of SBST with DSE by hooking together the evolutionary testing tool eToc [25] and the DSE tool jCUTE [22], such that either jCUTE optimizes the results of eToc, or vice versa. Such a sequential combination may not be sufficient to solve problems like the one illustrated in Figure 1, as several iterations of both techniques may be necessary. Majumdar and Sen [16] interleaved DSE with random search; DSE provides an exhaustive local search, while random search is used to explore more diverse parts of the state space.

The approach presented in this paper differs from these past approaches as it is applied in a scenario of where whole test suites are evolved, and DSE is in turn applied to the individuals of the GA population (which is related to the third category of DSE/SBST combinations). The approach is adaptive in choosing when to apply which technique, and allows for as many iterations as necessary. A further advantage of this type of integration over a simple sequential integration (e.g., [11]) is that it allows for optimization with respect to any coverage criterion or fitness function.

III. Integrating DSE in Whole Test Suite Generation

We consider a unit testing scenario, where the objective is to produce a test suite that is as small as possible, yet achieves highest possible coverage on the class under test (CUT) for a given coverage criterion. This is the target scenario of whole test suite generation, and in this section we consider how whole test suite generation can be extended with DSE to achieve higher coverage.

A. Whole Test Suite Generation

Whole test suite generation uses a GA to evolve a population of candidate solutions towards maximizing a given coverage criterion. Each individual of this population is a set T of test cases t_i; the size of a test suite is the number n of tests in the set. Each test case t_i consists of a sequence of calls on the CUT or its dependencies, in terms of their constructors, methods, fields, or creates primitive values and arrays [7]. Parameters of a call in the sequence are satisfied with values defined by earlier statements in the sequence.

Neither the number of tests in a test suite, nor the length of individual test cases is fixed, but may vary over time based on the search operators applied. When two individuals of the population are selected for crossover, then a random value α is chosen from $[0, 1]$, and the first offspring consists of the first $\alpha|P_1|$ test cases from the first parent P_1, followed by the last $(1-\alpha)|P_2|$ test cases from the second parent P_2; the second offspring is created from the other two subsets. When a test suite is mutated, then each test case is mutated with probability $1/n$, and new test cases are added with decreasing probability (initial probability σ, if a test is inserted then another one is inserted with probability σ^2, and so on). Test cases are deleted from test suites when they have length 0. Test cases are mutated using a range of operators that insert, remove, or change the statements in a sequence (see [7] for details). The initial population of the GA is created with test suites of random size, consisting of random test cases created by repeatedly applying the insertion mutation.

The GA is guided by a fitness function that represents the chosen coverage criterion. For example, the fitness function for branch coverage considers all methods and their conditional statements in the CUT. The branch distance for the evaluation of a given conditional statement to a particular truth value is an estimate of the distance towards this evaluation, and is an established heuristic in SBST [18]. The larger the branch distance, the "further away" a given execution was from making the branch evaluate to the chosen truth value; if the chosen branch evaluates to the target value then the branch distance is 0. The optimization goal of whole test suite generation is to create a test suite that covers all branches, therefore the fitness function considers the minimal branch distance for every single branch evaluation. If all branches are covered, then for each branch there has to exist a test case that achieves branch distance 0. Let $d_{min}(b, T)$ be the minimal branch distance for branch b on test suite T, then we define the distance $d(b, T)$ as follows:

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ \nu(d_{min}(b,T)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases}$$

The function $\nu = x/(x+1)$ normalizes the branch distance in the range $[0, 1]$ in order to prevent that any of the branches dominates the search [1]. The requirement to execute each branch twice is used to prevent the search from oscillating

between optimizing a branch between true and false evaluation. Considering that some of the methods M of the CUT may have no conditional statements, we also include the set of methods executed by test suite T as M_T, which results in the following fitness function [7]:

$$\text{fitness}(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k, T)$$

B. Dynamic Symbolic Execution of Test Suites

Traditional DSE assumes there is one entry point for which it should generate inputs to cover all execution paths, and the exploration starts with a single random input and successively solves variants of the resulting path condition in order to systematically explore all paths. The unit testing scenario considered in this paper is slightly different: There is not a single entry function, but all methods of a given CUT need to be exercised. Exploring all possible paths with DSE is not immediately possible in this scenario, because the search space is defined by the possible sets of method sequences (test suites), which cannot be explored with a constraint solver. Consequently, when integrating DSE into whole test suite generation the starting point for the DSE exploration is not a single random input, but an existing test suite.

We therefore consider all primitive values defined in the test cases of a test suite T as inputs on which the path conditions for DSE should be based. Consequently, every primitive statement (including definitions of string values) in T represents a *concrete* as well as a *symbolic* value. We can think of this as translating the concrete test cases from the test suite into parametrized unit tests [24].

In sequences of method calls individual primitive variables are often reused at different places, and so it can easily happen that the resulting constraints are formulated in such a way that negation leads to infeasible constraints. For example, assume we have the following function:

```
boolean same(int a, int b) {
  if(a == b) return true;
  return false;
}
```

If this method is called such that the same variable is used for both inputs, e.g., `same(x, x)`, then the path condition will claim that $x == x$, but the negation $x \neq x$ is not satisfiable. To avoid this problem, we *expand* test cases before collecting constraints. This simply means that each *occurrence* of a variable is considered an independent input (e.g., the function would now be called by `same(x1, x2)`, where $x1$ and $x2$ are both initialized with the same concrete value as x originally.

A primitive statement may also define an array, in which case each element of the array can be assigned a value through a dedicated assignment statement. When expanding a test case, we add an explicit assignment of the default value (e.g., 0) for every array slot unless it already has an assignment. This way, each array value will be considered as a symbolic input to the test case.

After expanding all test cases of a test suite, each of the tests is executed concolically considering all primitive values as symbolic input values. The resulting set of path condition serves as starting point for a regular DSE exploration: We select one branch condition at a time, negate it, and the conjunction of the reaching condition and the negated branch condition represent the input to a solver. A solution produced by the solver represents a new test case, which is executed concolically again, and the set of path conditions is updated for the next DSE step. This process is continued until a designated timeout for the DSE exploration has been reached or there are no more conditions to negate.

During this DSE exploration there are a number of parameters that need to be considered: First, there is the strategy to select path conditions for negation. Many different strategies have been considered in the literature (e.g., depth-first search, Fitnex [27], etc), and all of them could be applied at this point in the approach. Given the fixed and usually low time limit, we rank path conditions by their size, such that the exploration first considers those conditions that are likely easier to solve. In most cases this resembles a breadth-first exploration, but as a test case may contain several instances of the CUT this strategy is not related to the actual positions in the test cases. If the number of path conditions is high, then it is possible to optimise by considering only those that, for a given test suite, only evaluate to either true or to false, but not to both.

C. A Hybrid Adaptive Approach

Both, whole test suite generation and DSE, have been shown to be effective in many papers. However, there remains the question of how to best integrate the two approaches. When a CUT has many branches that depend on numerical constraints on the inputs, then DSE is clearly at an advantage. In the GA, each primitive input value in a test suite only has a small probability of being mutated, and there possibly have to be many mutations before a primitive value has been optimised towards covering a branch; after each step of mutations the fitness of the mutated individual needs to be re-evaluated, which requires re-executing the test suite and can be costly. On the other hand, if the challenge in covering the CUT lies in creating complex data types, then time spent on DSE should rather be invested into performing search, as performing DSE is also costly: First, there is the effort of executing all tests in the test suite *concolically* to collect the path conditions, and then there is the effort of negating branches and solving the resulting constraint systems. If the wrong technique is applied to the problem at hand and computational resources are bounded (e.g., timeout), then the effects on the achieved coverage can be detrimental.

Consequently, the integration of DSE and whole test suite generation requires three decisions: 1) On which individuals of the population is DSE applied, 2) when is it applied, and 3) how is it applied. If computational resources were not bounded, one would ideally apply DSE on every individual of the population at every iteration, essentially creating a form of *memetic algorithm* where individuals can improve themselves.

However, the ideal scenario of unlimited resources does not hold in practice, and thus applying DSE on all individuals at every iteration of the GA is out of question. A more conservative approach is therefore to apply DSE only to the *best* individual of the search. To reduce the computational costs, an obvious optimisation is to restrict DSE to only those cases where the best individual has changed from the previous generation (i.e., if there was mutation or crossover).

However, on a CUT where DSE is not helpful (e.g., when primitive input values do not determine many branches) even applying DSE only on the best individual after a change may have a negative impact on the search. To determine whether a CUT is suitable for DSE we therefore consider not only whether an individual has changed, but also what changes were applied and what were their consequences: If an individual was changed but the fitness has not been affected, then it is unlikely that DSE can improve the individual further. If an individual has improved fitness after changes that are not related to primitive values, then again it is unlikely that DSE can lead to an improvement. Consequently, we *adaptively* apply DSE: If we observe a change in fitness after a mutation on a primitive value, then we know that the variable this value is assigned to is important, and we can use DSE to derive new values for it.

Theoretically, a mutation step on a test suite can involve more than one mutation on its test cases, yet fitness is only evaluated after mutation of the entire test suite is complete. If there are several mutations, then it is not possible to know which of the mutations caused the fitness change. However, this is not a problem in practice, as the number of mutations is usually low. Given a test case of length l where each function call is mutated with probability $1/l$ [7], then on average $l \times 1/l = 1$ mutations will be applied. The probability $P(k)$ of having k mutations will be characterized by the following formula:

$$P(k) = \left(\frac{1}{l}\right)^k \times \left(1 - \frac{1}{l}\right)^{l-k} \times \binom{l}{k},$$

which means $P(0) = 0.34$, $P(1) = 0.38$, $P(2) = 0.19$, $P(3) = 0.05$, etc. Consequently, in practice it is acceptable to assume that a fitness change and a primitive mutation are linked.

Limiting the application of DSE to those cases where mutation of a primitive value has been shown to influence fitness will prevent unnecessary applications of DSE in most cases. However, even if a primitive value influences the fitness, it may be the case that the CUT results in path conditions that are too difficult for the solver to handle. For example, this can be the case if there are calls to external code that is not susceptible to DSE (e.g., native code in Java or web services), or if there are many infeasible branches, many loop iterations, or simply difficult path conditions). If DSE cannot be applied successfully, then one would want to minimize the amount of time spent on DSE, whereas in cases where DSE is successful one would want to make sure that DSE is applied. To achieve this, we can resort to *parameter control*: Whenever we observe that a primitive mutation has affected fitness, we apply DSE with a certain probability P. If this application of DSE was unsuccessful, we change P with a factor R, e.g., setting it to P/R. On the other hand, if it was successful then we can increase the probability to $P \times R$ (while keeping P in $(0, 1]$).

Finally, to restrict the DSE steps applied and the number of tests added to a test suite, we can evaluate the fitness of the test suite each time a new test case was added, and only keep the test in the test suite and for DSE exploration if it improves fitness. This would likely leads to smaller test suites and shorter DSE cycles, but may lead to removal of important tests in cases where the fitness function is not sufficiently fine grained enough (e.g., inter-procedural calls to other classes may not be rewarded by the fitness function).

IV. Evaluation

In this paper, we carried out a series of experiments to answer the following research questions:

RQ1: What is the best way to integrate DSE into a GA?

RQ2: What improvements does the integration of DSE achieve over a standard GA?

RQ3: How do the improvements vary over time?

RQ4: What are the actual benefits of integrating DSE on real-world software?

RQ5: How does the hybrid algorithm compare to regular DSE?

A. Empirical Setup

We have implemented DSE in EvoSuite based on the bytecode instrumentation implemented in the Dsc tool [12][2] as both tools are based on the same bytecode instrumentation framework. We implemented our own constraint representation in order to capture numeric as well as string constraints. To cope with such mixed constraints, we implemented our own constraint solver, which we can only briefly describe for reasons of space. The solver applies the Alternating Variable Method (AVM) [14]. Integer numbers are solved using standard exploratory and pattern moves, whereas for floating point variables we iterate over precision values as proposed by Harman and McMinn [10] and also implemented in the Flopsy [15] tool. Finally, to solve string constraints we interpret a string variable as an array of characters and apply AVM on each character in turn. It is likely that dedicated solvers such as Z3 [5] or Hampi [13] could further improve performance, at least for numeric datatypes. However, to measure improvement over a pure search-based whole test suite generation approach this is not necessary. Internally in EvoSuite, a test case can be executed for DSE or for regular SBST simply by choosing the corresponding classloader. We carried out a series of experiments using three different cases studies, which are described in detail below.

Both GA and DSE are based on randomized algorithms. To properly compare these algorithms, all data resulting from the empirical analysis were analyzed using statistical methods following the guidelines in [2]. In particular, we used the

[2]Available at http://ranger.uta.edu/˜csallner/dsc

Vargha-Delaney $\hat{A}_{12}$ effect size and Wilcox-Mann-Whitney U-test. This test is used when algorithms (e.g., result data sets X and Y) are compared on single classes (in R this is done with $wilcox.test(X, Y)$). We also used this test to check on the entire case study if effect sizes are symmetric around 0.5. On some classes, an algorithm can be better than another one (i.e., $\hat{A}_{12} > 0.5$), but on other classes it can be worse (i.e., $\hat{A}_{12} < 0.5$). A test for symmetry determines if there are as many classes in which we get better results as there are classes in which we get worse results. Statistical tests are carried out at $\alpha = 0.05$ significance level.

B. Best Configuration for the Integration

The first question we need to consider before performing in-depth analysis is how to combine DSE and SBST. In the past, we have experimented with using DSE as a mutation operator for SBST [17], but in a unit testing approach with sequences of method calls we wanted to have a more focused approach that reduces the amount of costly DSE executions. A loose coupling as implemented in the EVACON tool [11] would not be able to handle cases like Figure 1, as this requires iterations of both tools, rather than just successive application[3] Our initial attempts of applying DSE at a fixed rate (e.g., every X generations) showed a detrimental effect on the achieved coverage, except in extreme cases that are trivial for DSE and difficult for SBST. We therefore turned to our adaptive approach as described in the previous section, but there still remain several parameters to decide on:

1) How much time should be spent on DSE?
2) Should only test cases that improve coverage be retained, or all tests produced by DSE?
3) Should the set of branch conditions be filtered according to which branches are only covered one way?
4) Should DSE be applied whenever an individual changed its fitness after a primitive mutation, or only with a certain probability?
5) Should this probability be adaptively updated depending on the success of DSE?

To answer these questions, our first case study is composed of 38 classes, which represent classes where SBST and EvoSuite struggle. Of the 38 classes, 22 are taken from a recent case study on complex string problems in SBST [19]. The remaining 16 classes are chosen based on our past experiments and all represent cases where EvoSuite has problems in achieving high coverage, even though achieving coverage on these classes is not inhibited by environmental dependencies. Table I summarizes statistics of these classes (the first 22 classes are from [19]).

To answer **RQ1**, we considered the following values for the above listed parameters:

1) Time: 5s, 10s, 30s
2) Keep all tests: true, false
3) Filter branch conditions: true, false
4) Probability: 0.1, 0.5, 1.0

[3]Note also that EVACON is not available online for comparison.

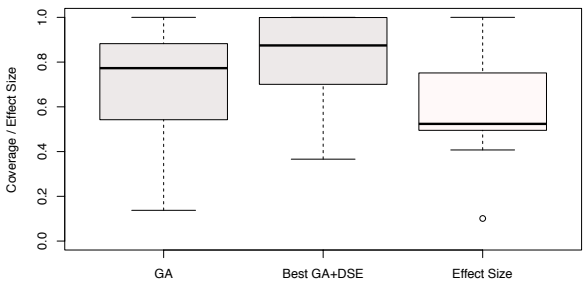

Fig. 2. Average branch coverage achieved by EvoSuite using only the GA, and using the hybrid approach with DSE and the best configuration, as determined for RQ1.

5) Adaptation rate: 1, 2, 4

In addition, we included four configurations of applying DSE on all individuals if they showed improvement after a primitive mutation, instead of only the best individual. In total, this led to 112 different configurations of DSE. All these configurations (and the basic GA) were run on each of the 38 classes in the first case study, each with a timeout of 10 minutes. To take the randomness of these algorithms into account, each experiment was repeated 16 times with different random seeds. In total, this first set of experiments took $(113 \times 38 \times 16 \times 10)/(60 \times 24) = 477$ days of computational time. To identify which of the 112 configurations can be considered as best, for each configuration we calculated the effect size on each class compared to the standard GA. The best configuration was identified as the one with highest average effect size.

RQ1*: The best configuration in our experiments is:*
1) Time: 30s
2) Keep all tests: false
3) Filter branch conditions: true
4) Probability: 1.0
5) Adaptation rate: 1

Consequently, the best results on the case study examples were achieved with applying DSE always for a long time when the best individual improved, and without adapting the probability. Only tests that improved the fitness were retained, and only branches that are only partially covered were negated.

1) Comparison to GA: Based on the best configuration for the combination of the GA and DSE, we investigated the achieved improvement in detail (**RQ2**). To obtain more statistical power, we ran more experiments with these two configurations, i.e., 100 times (instead of just 16) on each of the 38 classes. Figure 2 shows boxplots of the achieved coverage values for EvoSuite using only the GA, and for EvoSuite using the GA and DSE. On average, the GA achieved 71.2% branch coverage, whereas the combination with DSE increased the average coverage to 82.2%. On 15 out of the 38 classes the increase in coverage is statistically significant, whereas on 2 classes there was a statistically significant decrease; in the other classes there is no significant difference. The average effect size $\hat{A}_{12}$ for the comparison of

TABLE I
DETAILS OF THE CLASSES USED IN THE FIRST CASE STUDY.

LOC are lines of non-commenting source code measured by JavaNCSS (http://www.kclee.de/clemens/java/javancss/); Branches are measure by EVOSUITE and are measured at bytecode level.

Project	Class	LOC	Branches
Chemeval (chemeval.sf.net)	org.openscience.cdk.index.CASNumber	34	15
Conzilla (www.conzilla.org)	se.kth.cid.identity.ResourceURL	22	12
	se.kth.cid.identity.URI	83	54
	se.kth.cid.identity.URIClassifier	52	14
	se.kth.cid.identity.URIUtil	39	13
	se.kth.cid.identity.PathURN	17	10
	se.kth.cid.identity.URN	17	12
	se.kth.cid.identity.MIMEType	32	12
Efisto (efisto.sf.net)	com.efisto.util.Util	74	20
GSV05 (gsv05.sf.net)	stempeluhr.validation.TimeChecker	33	11
JXPFW (jxpfw.sf.net)	org.jxpfw.util.InternationalBankAccountNumber	110	54
	org.jxpfw.util.CLocale	18	10
LGOL (lgol.sf.net)	uk.gov.tameside.apps.validation.NumericValidator	16	8
	uk.gov.tameside.apps.validation.EmailValidator	17	8
	uk.gov.tameside.apps.validation.PostCodeValidator	44	12
	uk.gov.tameside.apps.validation.TelephoneValidator	17	8
	uk.gov.tameside.apps.validation.DateFormatValidator	20	10
OpenSymphony (www.opensymphony.com)	webwork.examples.userreg.Validator	46	42
PuzzleBazar (code.google.com/p/puzzlebaazar)	com.puzzlebazar.client.util.Validation	48	97
WIFE (wife.sf.net)	com.prowidesoftware.swift.model.IBAN	62	27
	com.prowidesoftware.swift.model.BIC	24	13
Java Naming and Directory Interface (JNDI)	com.sun.jndi.toolkit.url.ConcreteURLContext	227	65
Roops (http://code.google.com/p/roops/)	roops.core.bv32.linear.noex.gods.LinearWithoutOverflow	223	93
	roops.extended.bv32.floats.FloatArithmetic	68	49
String case study [7]	Cookie	17	13
	DateParse	31	39
Numerical case study [7]	Remainder	32	25
	Bessj	79	29
Commons CLI (commons.apache.org/cli)	org.apache.commons.cli.CommandLine	87	45
JDom (www.jdom.org)	org.jdom.Attribute	138	65
Commons Codec (commons.apache.org/codec)	org.apache.commons.codec.language.DoubleMetaphone	579	504
Java	java.util.ArrayList	151	70
JodaTime (joda-time.sourceforge.net)	org.joda.time.DateTime	339	148
	org.joda.time.format.DateTimeFormat	356	434
JGraphT (jgrapht.org)	org.jgrapht.alg.BellmanFordIterator	105	42
Commons Math (commons.apache.org/math)	org.apache.commons.math.transform.FastFourierTransformer	290	135
Java	java.util.regex.Pattern	2,701	1,743
NanoXML (nanoxml.sourceforge.net/orig)	net.n3.nanoxml.XMLElement	661	310

the DSE version with the standard GA is 0.62, such that we can conclude that the integration of DSE into the GA leads to an improvement with strong statistical significance.

Table II lists the results in detail per class. The two classes where the hybrid approach is worse are *DoubleMetaphone* and *DateTime*. For *DoubleMetaphone* the average coverage achieved by the hybrid approach is actually higher, but there is high variance on this class for the chosen configuration. Considering the results on this class over all configurations, we see that a better configuration for this particular class would use only a 50% probability of applying DSE and adapts this rate with a factor of 2. This configuration leads to an average coverage of 71.3%, and the improvement over GA is statistically significant with $\hat{A}_{12} = 0.61$. For *DateTime* the best configuration also uses an adaptation rate of 2, but with a starting probability of 10% and a significantly lower timeout for DSE of 5 seconds. With this configuration the hybrid approach achieves an average coverage of 84% (maximum of 98.2%) with a small effect size of $\hat{A}_{12} = 0.51$. Indeed, every single class in the benchmark has a configuration where the hybrid approach is better than the GA, which suggests that even though we have identified a configuration that achieves significantly better overall results, there is still potential to refine the adaptiveness of the configuration in future work.

RQ2*: The hybrid approach increased the average branch coverage from 71% to 82%.*

2) Improvements over Time: Ten minutes can be a long time for test generation, and it may be the case that the improvement of DSE only appears after spending a certain amount of time, whereas smaller time budgets might show a negative effect; in particular, the best configuration was chosen out of runs of 10 minutes. To investigate the effects over time, for **RQ3** we kept track of achieved coverage at each minute interval for the configuration chosen in **RQ1**. Figure 3 clearly shows that the beneficial effect of DSE does not only appear over time, but applies from the beginning.

RQ3*: The integration of DSE is beneficial independently of the applied search budget.*

TABLE II
AVERAGE COVERAGE PER CLASSES IN THE FIRST CASE STUDY AND EFFECT SIZE OF THE HYBRID APPROACH OVER THE GA.

$\hat{A}_{12} < 0.5$ means the hybrid approach resulted in lower, $\hat{A}_{12} = 0.5$ equal, and $\hat{A}_{12} > 0.5$ higher coverage than the standard GA. Significance at 0.05 level is shown with bold font.

Class	GA	GA+DSE	$\hat{A}_{12}$
CASNumber	0.38	0.37	0.48
ResourceURL	0.94	1.00	**0.66**
URI	0.81	0.91	**0.94**
URIClassifier	0.81	0.93	**0.75**
URIUtil	1.00	1.00	0.50
PathURN	0.23	0.52	**0.66**
URN	0.89	0.97	**0.62**
MIMEType	0.78	0.85	0.55
Util	1.00	1.00	0.49
TimeChecker	0.62	0.61	0.47
InternationalBankAccountNumber	0.70	0.80	**0.89**
CLocale	1.00	1.00	0.50
NumericValidator	1.00	1.00	0.50
EmailValidator	1.00	1.00	0.50
PostCodeValidator	1.00	1.00	0.50
TelephoneValidator	1.00	1.00	0.50
DateFormatValidator	0.81	0.78	0.48
Validator	0.38	0.40	0.53
Validation	0.76	0.92	**0.84**
IBAN	0.63	0.68	0.54
BIC	0.78	0.87	**0.55**
ConcreteURLContext	0.54	0.83	**0.88**
LinearWithoutOverflow	0.38	0.98	**0.99**
FloatArithmetic	0.41	0.85	**0.99**
Cookie	0.45	1.00	**1.00**
DateParse	0.67	1.00	**1.00**
Remainder	0.88	0.85	0.48
Bessj	0.87	0.90	0.52
CommandLine	0.88	0.90	0.52
Attribute	0.74	0.83	**0.94**
DoubleMetaphone	0.45	0.69	**0.41**
ArrayList	0.81	0.88	**0.61**
DateTime	0.87	0.56	**0.10**
DateTimeFormat	0.44	0.76	0.48
BellmanFordIterator	0.66	0.70	0.51
FastFourierTransformer	0.58	0.59	0.47
XMLElement	0.75	0.95	0.46
Pattern	0.14	0.37	0.70

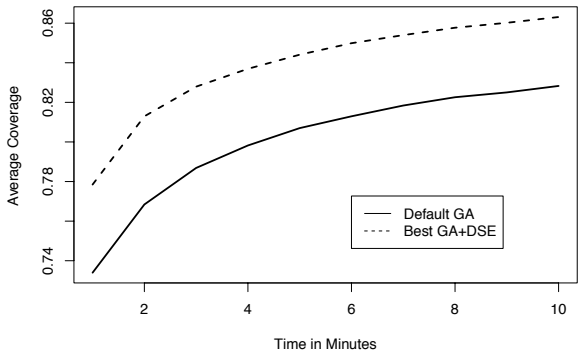

Fig. 3. Branch coverage results over time.

3) Evaluation on SF100: All experiments so far were performed on the set of 38 classes representing instances of difficult problems from an SBST point of view, such that DSE leads to clear improvements. Consequently, we are confident that DSE can improve SBST when the problem at hand is suitable. However, not all classes actually exhibit the necessary properties; potentially, complex numerical and string constraints may even be rare in practice. To see whether our results generalize and whether the integration has a negative effect on the general use of EVOSUITE, we applied the combined approach on all 8,784 classes of the SF100 corpus of classes [6]. SF100 consists of 100 randomly chosen open source projects from Sourceforge, and therefore represents a statistically valid sample of open source software. Because of the large number of classes, we reduced the timeout of EVOSUITE to two minutes when testing SF100, and experiments were repeated with only five different random seeds. In total, these experiments required $(2 \times 8784 \times 5 \times 2)/(60 \times 24) = 122$ days of computational time.

The standard GA obtained 55.2% branch coverage, whereas DSE obtained 56.4%, and the effect size is 0.515. A test for symmetry around 0.5 shows a very strong statistical difference, i.e., p-value very close to zero. If we look at each class in isolation, there are 2,397 cases in which $\hat{A}_{12} > 0.5$ (i.e., better results), and 1,782 in which $\hat{A}_{12} < 0.5$ (i.e., worse results). On each class in isolation, there are 124 cases in which $\hat{A}_{12} > 0.5$ and the U-test gives a p-value lower than $\alpha = 0.05$, whereas there are only nine cases in which we obtain such small p-values for $\hat{A}_{12} < 0.5$.

All these data can be interpreted as follows: there is strong statistical evidence to claim that, on SF100, the hybrid GA+DSE is better than GA. But, on the other hand, five runs per class are not enough to obtain a strong enough statistical power to precisely identify on which classes DSE is indeed better.

Compared to the results on the previous case study, the improvements of our hybrid approach are lower, i.e. $56.4 - 55.2 = 1.2\%$ compared to $82.2 - 71.2 = 11\%$. This is expected, as the SF100 corpus features several real-world problems that are out of the reach of current test data generation tools, like for example environment dependencies (e.g., test data coming from writing/reading files and opening TCP connections).

RQ4*: Our experiments show with statistical significance that the integration of SBST and DSE is beneficial in practice, and has no significant negative effect.*

4) Comparison to DSE: To compare our approach to a standard DSE tool neither of our two sets of classes is suitable, as all DSE tools for Java we are aware of require a dedicated entry method for test generation, and have only limited support to generate complex data types. We therefore chose the Roops set of benchmark classes for this experiment (see Table III for details), for which the main developer of DSC [12] is one of the main contributors. We chose DSC as the DSE tool to compare to, as the DSE implementation in EVOSUITE is based on DSC. Consequently, we assume that the set of classes in Roops is suitable for DSC and the tool is not put at a disadvantage in comparison. Because DSC still requires a dedicated entry method, we split all of the Roops classes into individual classes of one method each (all methods in Roops classes are static), and compare DSC and EVOSUITE

TABLE III
ROOPS BENCHMARK DETAILS.

M = Methods; Branches (B) are measure by EVOSUITE at bytecode level; LOC are lines of non-commenting source code measured by JavaNCSS (http://www.kclee.de/clemens/java/javancss/)

Class	M	LOC	B
collections.IntRedBlackTreeMap	13	259	103
core.bv32.arr.noex.IntArrayWithoutExceptions	7	62	43
core.bv32.linear.noex.gods.LinearWithoutOverflow	44	221	93
core.objects.AvlTree	15	119	74
core.objects.BinTree	17	189	99
core.objects.BinomialHeap	19	351	210
core.objects.FibHeap	10	194	94
core.objects.LinkedList	17	126	59
core.objects.NodeCachingLinkedList	24	225	106
core.objects.Objects	3	16	9
core.objects.SinglyLinkedList	8	98	47
core.objects.TreeSet	28	455	218
extended.bv32.DeepASTrandom	196	981	1,108
extended.bv32.arr.BinSearchError	2	22	11
extended.bv32.arr.noex.IntArrayWithoutExceptions	3	25	27
—.IntArrayWithoutExceptionsWithArrayParameters	3	28	29

TABLE IV
AVERAGE BRANCH COVERAGE PER METHOD PER CLASSES, AND EFFECT SIZE OF THE HYBRID APPROACH OVER DSC.

Class	GA	DSC	GA+DSE	$\hat{A}_{12}$
IntRedBlackTreeMap	0.57	0.03	0.57	**1.00**
IntArrayWithoutExceptions	0.88	0.88	0.88	0.50
LinearWithoutOverflow	0.93	1.00	1.00	**0.50**
AvlTree	0.73	0.057	0.72	**1.00**
BinomialHeap	0.54	0.033	0.47	**1.00**
BinTree	0.53	0.11	0.53	**1.00**
FibHeap	0.55	0.12	0.54	**1.00**
LinkedList	0.80	0.29	0.80	**1.00**
NodeCachingLinkedList	0.66	0.43	0.65	**0.88**
Objects	1.00	1.00	1.00	0.50
SinglyLinkedList	0.98	0.17	0.99	**1.00**
TreeSet	0.66	0.096	0.72	**1.00**
BinSearchError	0.80	0.20	0.80	**1.00**
IntArrayWithoutExceptions	0.92	0.00	0.94	**1.00**
IAWEWithArrayParameters	0.88	0.29	0.88	**1.00**
DeepASTrandom	0.90	0.92	0.92	0.49

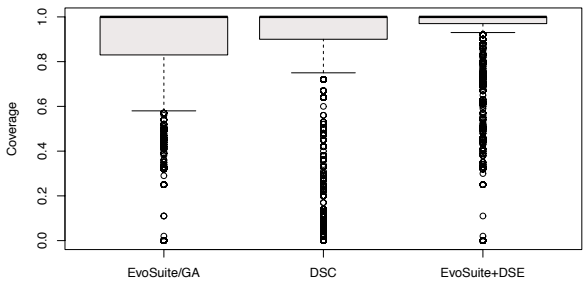

Fig. 4. Branch coverage results on Roops for EVOSUITE in standard search mode, DSC, and EVOSUITE with DSE.

on a per-method base. This setup allowed us to automatically run experiments with 10 repetitions to accommodate for the randomness of the tools, and to measure the coverage of the resulting JUnit test suites using EVOSUITE. EVOSUITE was used in the configuration chosen above. Both EVOSUITE and DSC were run with a time limit of one minute per class.

Figure 4 illustrates the results of applying EVOSUITE with and without DSE and DSC on the Roops benchmark. The figure shows that all three techniques are very close in coverage, and suggests that EVOSUITE with DSE achieves the best results. Interestingly, DSC achieved the overall lowest average coverage (84.1%), whereas EVOSUITE (87.9%) and EVOSUITE with DSE (91.3%) are both slightly better. Statistical analysis shows that the improvement of EVOSUITE +DSE over DSC is statistically significant, with $\hat{A}_{12} = 0.56$ (test for symmetry around 0.5 has p-value close to zero). In detail, there are 48 methods on which the combination is significantly better than DSC, whereas there are 14 on which it is worse; for the rest the approaches achieve the same coverage (which, as can be seen in Figure 4, is 100% in many of these cases). Table 4 summarizes the results, averaged per class. The table reveals that the high coverage values for DSC witnessed in Figure 4 are mainly due to a few classes with many methods (e.g., *DeepASTrandom*) where DSC works well, whereas there are many other classes where DSC has problems.

RQ5*: In our experiments,* EVOSUITE *with DSE achieved higher coverage than the DSE tool* DSC.

However, note that this result compares two tools, and comparisons between other tools implementing the same techniques may lead to different results. Consequently, we encourage reproduction of our experiments, and we are planning to compare to Pex [24] and other tools as future work.

C. Threats to Validity

The focus of this paper is on comparing the whole test suite generation approach based on a GA to a hybrid version that integrates a variant of DSE.

Threats to *construct validity* are on how we measured the performance of a testing technique. We used branch coverage, but in practice a small increase in code coverage might not be desirable if it implies a large increase of the test suite size. Furthermore, using only coverage does not take the human oracle costs into account, i.e., how difficult it will be to manually evaluate the test cases and to add assert statements.

Threats to *internal validity* might come from how the empirical study was carried out. We have carefully tested our framework to reduce the probability of having faults, but it is well known that testing alone cannot prove the absence of defects. To prevent that findings on a randomized algorithm are affected by chance we repeated each experiment a number of times and followed rigorous statistical procedures to evaluate their results. Our implementation of DSE and our search-based solver are very basic, and not optimized for performance. This means that more mature DSE tools might be able to extract more constraints where our tool fails, and might spend less time on doing so; this could influence the optimal configuration, and it could increase the overall achieved coverage.

There is also the threat to *external validity* regarding the generalization to other types of software, which is common for any empirical analysis. Because of the large number of experiments required, we only used 38 classes for our main evaluation, and then applied only a restricted experiment to SF100. It may be that other configurations would achieve

better results on SF100. Recall that SF100 is a statistically valid sample of 100 open-source Java projects. Therefore, results on SF100 have high probability to generalize to other open-source software as well.

V. Conclusions

Generating test sets that cover all branches of a program is of practical importance for software testing as well as other dynamic software analysis techniques. In the context of unit testing for object-oriented software, tests are sequences of method calls which can be efficiently generated with Genetic Algorithms (GAs). Although this approach performs well at finding good sequences of method calls, it can struggle on problems for which dynamic symbolic execution (DSE) can be very efficient. To overcome this problem, we presented a hybrid approach that integrates a GA with DSE.

Finding good values for the many parameters that a hybrid approach, like the one in this paper, comes with is essential to determine the true value of the combination. We therefore applied a rigorous approach in our evaluation; this evaluation on different sets of case study classes demonstrated that this hybrid approach succeeds at improving the coverage.

Our initial set of experiments demonstrated that the hybrid approach can lead to an increased branch coverage. Yet, there remain several open questions that need to be addressed. First, our own DSE implementation cannot match the level of efficiency provided by state-of-the-art tools like Pex; likely, improvements on our DSE implementation would lead to further improvements of the overall performance. In particular, we used a custom-made solver for mixed constraints, which can likely be further optimized for some domains using existing solvers such as Z3 [5]. We focused on branch coverage in our evaluation, but one of the advantages of using search-based techniques is that any coverage criterion for which one can define a fitness function can be used. This as well as larger empirical studies will be part of our future work.

To learn more about EvoSuite, visit our Web site:

`http://www.evosuite.org`

Acknowledgments. This project has been funded by a Google Focused Research Award on "Test Amplification", the Norwegian Research Council, the ERC grant SPECMATE, and EU FP7 grant 295261 (MEALS). We thank Christoph Csallner for source access to Dsc, and Andreas Zeller and Jeremias Rössler for comments on an earlier version of the paper.

References

[1] A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability (STVR)*, 23(2):119–147, 2013.

[2] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability (STVR)*, 2012. (to appear).

[3] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2011.

[4] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. Pasareanu. Symbolic execution with interval solving and meta-heuristic search. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[5] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[6] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.

[7] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[8] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 28(2):278–292, 2012.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Conference on Programming language design and implementation (PLDI)*, pages 213–223, 2005.

[10] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 73–83, 2007.

[11] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE'08: Proc. of the 23rd IEEE/ACM Int. Conference on Automated Software Engineering*, pages 297–306, 2008.

[12] M. Islam and C. Csallner. Dsc+mock: A test case + mock class generator in support of coding against interfaces. In *International Workshop on Dynamic Analysis (WODA)*, pages 26–31, 2010.

[13] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *Proceedings of the 18th Int. Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116. ACM, 2009.

[14] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, pages 870–879, 1990.

[15] K. Lakhotia, N. Tillmann, M. Harman, and J. De Halleux. Flopsy: search-based floating point constraint solving for symbolic execution. In *Proceedings of the 22nd Int. Conference on Testing Software and Systems*, ICTSS'10, pages 142–157. Springer-Verlag, 2010.

[16] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[17] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2011.

[18] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[19] P. McMinn, M. Shahbaz, and M. Stevenson. Search-based test input generation for string data types using the results of web queries. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 141–150. IEEE, 2012.

[20] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[21] A. Sakti, Y. Guéhéneuc, and G. Pesant. Boosting search based testing by using constraint based testing. *Search Based Software Engineering*, pages 213–227, 2012.

[22] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proc. of the 10th European Software Engineering Conf. held jointly with 13th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, pages 263–272. ACM, 2005.

[23] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu. Coral: solving complex constraints for symbolic pathfinder. In *Proc. Conference on NASA Formal methods*, NFM'11, pages 359–374, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] N. Tillmann and N. J. de Halleux. Pex — white box test generation for .NET. In *International Conference on Tests And Proofs (TAP)*, pages 134–253, 2008.

[25] P. Tonella. Evolutionary testing of classes. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[26] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proc. Int. Conference on Software Engineering*, ICSE '11, pages 611–620. ACM, 2011.

[27] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 359–368. IEEE, 2009.