An Improved Memetic Algorithm with Method Dependence Relations (MAMDR)

Ali Aburas
Oregon State University
aburasa@onid.orst.edu

Alex Groce
Oregon State University
agroce@gmail.com

ABSTRACT

Search-based approaches are successfully used for generating unit tests for object-oriented programs in Java. However, these approaches may struggle to generate sequence method calls with specific values to achieve high coverage due to the large size of the search space. This paper proposes a memetic algorithm (MA) approach in which static analysis is used to identify method dependence relations (MDR) based on the field access. This method dependence information is employed for reducing the search space and used to guide the search towards regions that lead to full (or at least high) structural coverage.

Our approach, MAMDR, combines both a genetic algorithm (GA) and Hill Climbing (HC) to generate test data for Java programs. The former is used to produce test cases that maximize the branch coverage of the CUT, while minimizing the length of each test case. The latter is used to target uncovered branches in the preceding search phase using *static* information that guides the search to generate sequences of method calls and values that could cover target branches. We compare MAMDR with pure random testing, a well-known search based approach (EvoSuite), and a simple MA on several open source projects and classes, and show that the combination of MA and MDR is effective.

Keywords

Search Based Software Testing, Memetic Algorithms, Static Analysis, Search Space Reduction, Object-Oriented.

1. INTRODUCTION

Achieving high coverage in object-oriented programs like Java is a very challenging and expensive task. Creating a unit test to achieve a high structural coverage, e.g. branch coverage, of a class under test (CUT) requires a desirable sequence of method calls that create and put objects into particular states. These objects can be used as the receiver or arguments of the method(s) under test (MUT). When automatically performing unit test generation, the primary goal is to ensure that all, or at least a large number of the control statements in the CUT are executed, which gains confidence in the CUT's quality and functionality.

There are many automated test generation approaches, including random testing, symbolic execution-based, and search-based approaches. *Random testing* approaches [11, 29] are easy to implement, applicable, and the fastest in execution [42]. Despite the advantages that random testing provides, it is still considered weak for achieving high structural coverage. The main reason for low coverage is that random testing faces a challenge in producing a sequence of method calls with specific arguments for complex programs. Approaches based on *symbolic executions* (e.g., KLEE [10]) explore path conditions in the program under test and collect constraints on all inputs from the branch statements. If the collected constraints are feasible, then a constraint solver is used to generate input from them. However, these approaches face a challenge of scalability if the program under test is complex. Search-based test generation approaches (e.g., EvoSuite [13]) have already been shown to be effective for generating test data that achieves high code coverage and reveals failures [8, 13]. The *search-based* approaches consider more than one solution at the same time. They employ meta-heuristic optimization techniques, such as Genetic Algorithms, and use a fitness function that guides the search toward better solutions. However, in particular circumstances, these approaches face challenges which negatively affect their ability to achieve high structural coverage for certain programs. When we have a large number of methods to test, each of which can take some parameters as inputs, then finding the potential methods calls to optimize the solutions can be a challenge due to the large size of the search space [6, 16]. In addition, the efficiency of the search-based approaches decreases as well for programs that have predicates using string constants. In this case, no heuristic can be defined to guide the search, since the fitness function landscape contains plateaus [6, 28].

One potential way to alleviate these problems, and improve the effectiveness of the search-based approaches is to use static analysis and exploit the program under test to guide search-based test generation. This was the main motivation for this work: we focus on developing a search-based approach which generates test inputs for object-oriented programs and utilizes the source code of the program under test to overcome the aforementioned challenges.

This paper presents an automated search-based technique that uses dependence relations among the MUT based on the fields they access [42]. The goal of this technique is to guide the search to look in the most promising regions of the search space by eliminating irrelevant method calls without reducing the search effectiveness. Fraser and Arcuri [14] conducted a study on 20 Java projects, concluding that the use of seeding can strongly improve performance of an evolutionary search. Consequently, we introduce seeding that uses constants provided by the source code of the program under test to help the search cover certain branches that are dependent on particular values.

The main contributions of this paper are the following:

1. We introduce a search-based approach to automatic test generation based on memetic algorithms. We extend global search (Genetic Algorithm) with a local search (Hill Climbing).
2. We introduce a technique to reduce the search space for object oriented programs, based on method dependence relations [42].
3. We also introduce a way to seed constants into the search process when targeting uncovered branches.
4. We present the results of an empirical study on 4 popular open source programs and 6 Java classes. Some of these classes are taken from recent experiments where search-based approaches, like EvoSuite, struggled with challenges in achieving high coverage. The results show the effectiveness and impact of our approach.

This paper is organized as follows. Section 2 introduces the concept of our approach with an illustrative example. Section 3 provides background information, and related work is reviewed in

Section 4. Section 5, describes in details our approach. Section 6 discusses the evaluation setup of the empirical study. Section 7 the results of the evaluations of the approach are presented and discussed. Threats of validity are analyzed in Section 8. Finally, conclusions and future work in Section 9.

2. MOTIVATING EXAMPLE

In this section, we illustrate some of the issues involved the search process, through an illustrative example taken from the NanoXML[1] [16] project, shown in Figure 1.

Figure 1 shows one class under test (CUT), `CDATAReader`, and we consider a method `read` as a method under test (MUT), which returns the number of characters read, or -1 if at EOF. For simplicity, we did not show constructors of the class `StdXMLReader`.

Creating the desired object states of the receiver or arguments of a MUT is required to achieve full or at least high coverage in the MUT. For example, creating a `CDATAReader` object of the method under test involves the creation of non-primitive parameters at line 5. Therefore, the `CDATAReader` object and the parameter `StdXMLReader` (a concrete implementation of the interface class `IXMLReader`) object must be in desired states to cover particular branches. Moreover, the `read` method contains some branches that require a particular character value, such as ']', at line 18. If the size of the test cluster consists of a large number of classes and public methods, the search will struggle to randomly pick the right methods and arguments as candidates that help to cover the required branches. In fact, some branches predicates involve a Boolean value, such as B3 at line 19, i.e. the flag problem [28]. As a result, no heuristic can be defined that gives guidance on how to cover the target branch B3. In such cases, the search space will have large plateaus and the search will likely degenerate to pure randomness, since no information can be exploited to guide the search on how to change the flow of the execution [28].

```
1.  class CDATAReader extends Reader {
2.  private IXMLReader reader;
3.  private char savedChar;
4.  private boolean atEndOfData;
5.  CDATAReader(IXMLReader reader){
6.        this.reader = reader;
7.        this.savedChar = 0;
8.        this.atEndOfData = false;
9.  }
10. public int read(…)throws IOException {
11. …
12.   while (…) {
13.     Char ch =this.savedChar;
14.     if (ch == 0)
15.        ch = this.reader.read();//B1
16.     Else
17.        this.savedChar = 0;  //B2
18.     if (ch == ']') {
19.       char ch2 = this.reader.read(); //B3
20.        if (ch2 == ']')
21.      … more if statements …
22.      }
23.   }…
24. }
25. … 3 more methods …
26. }
27. public class StdXMLReader  implements IXMLReader{
28. …
29. public static IXMLReader stringReader(String str){
30.    return new StdXMLReader(new StringReader(str));
31. }
32. … 20 more methods …
33. }
```

Figure 1: Two classes taken from the NanoXML project.

[1] http://nanoxml.sourceforge.net/orig/

Searching through regions of the search space that do not produce a desired object state will increase the number of fitness function evaluations without any gain in covering useful code [6]. As revealed by our experimental results in section 6, pure random testing, a search-based approach (EvoSuite), and a simple MA [6, 8] could achieve 69%, 68% and 77% branch coverage of the `CDATAReader` class, respectively.

Our approach intelligently reduces and navigates the search space and recommends candidate methods or constructors that help to cover a target branch. The space search reduction approach used in this paper is based on the concept of Method Dependence Relations (MDR) [42]. We use static analysis to analyze a target branch predicate and identify the relevant member fields and/or parameters of MUT which will be responsible for covering the target branch. Removing irrelevant inputs can improve search performance. Furthermore, constant primitive values (e.g. numbers or strings) are extracted from target branches, and preferred over randomly generating new values.

MAMDR uses two phases of static analysis to identify relations between methods. In the first phase, it statically identifies method dependence relations based on the read and written fields and then recommends all the public methods and constructors that write a particular field. In the second phase, the signatures of each recommended public method and constructor are analyzed; afterwards, all constructors that create instances and methods that return the same data type of the non-primitive parameters are added to the recommended list. For example, consider covering branch B3 at line 19 (Figure 1). A necessary requirement to cover branch B3 is that non-primitive field `reader` must contain character value ']'. Consequently, MAMDR recommends the constructor of `CDATAReader` that writes the field `reader`. Then, MAMDR also recommends both the class constructor of `StdXMLReader`, and method `stringReader` at line 27, because they both return instances that can be used to replace the interface class type argument in the `CDATAReader` constructor. Finally, MAMDR uses the character constant ']' to initialize the inputs of the arguments instead of randomly initializing them. For instance, if MAMDR picks the method `stringReader` to invoke, then the parameter of string type at line 27 is initiated with the character value ']'. This combined MDR and branch predicates constants extraction information allows MAMDR to generate more effective sequences of method calls that cover branches that require specific input values. Our results show that MAMDR achieves 96% branch coverage of the CDATAReader, which is 27% higher than pure random testing, 28% higher than EvoSuite and 19% higher than a simple MA.

3. BACKGROUND

In this section we describe some Search-Based Software Testing (SBST) algorithms that have been applied in software test data generation.

3.1 *Evolutionary Algorithms*

Evolutionary algorithms [28] are based on the idea of genetics and evolution in which new and fitter sets of candidate solutions, which are often called individuals or chromosomes, are created by combining portions of fittest candidate solutions. Genetic Algorithms (GA) are probably the most common technique in Evolutionary Algorithms [28]. GA starts with a random initial population of individuals. Then, the algorithm enters evolutionary iterations with the following order: First, each individual is executed and its fitness is computed. Second, individuals based on their fitness are selected. Then, a recombination operator is applied by taking two parent individuals and producing two new offspring.

After recombination, a mutation is applied, which produces small random changes to the offspring. Finally, these new offspring fill the population of the next generation. The evolution is performed until a termination criterion is met, for example time budget or number of generations. To avoid the possible loss of the fittest individuals (elitism), the new population is always initialized with a number of best individuals without any modification.

The individual length, population size, and the crossover and mutation probabilities values in GA are referred to as GA parameters. In addition, selection, crossover, and mutation are referred to as GA operators.

There is also a subset of genetic algorithms [31], called Genetic Programming (GP), and sharing many characteristics with GA, such as the operators of selection, reproduction, and mutation. However, the difference between the two is the representation of the individuals: in GP the individual is normally represented as a tree-structure.

3.2 *Local Search Algorithms*

In contrast to GA, local search algorithms aim to improve one individual by exploring its neighbors [28]. Hill Climbing (HC) is a well known local search algorithm. It usually starts with a random individual, and then it considers the set of near neighbors to this individual. If a fitter neighbor is found, HC moves to it and again it investigates its neighbors. If HC gets trapped in a local optimum, which there is no better neighbor is found, it randomly restarts from a new individual.

3.3 *Alternating Variable Method (AVM)*

The Alternating Variable Method (AVM) is a similar technique to HC, and developed by Korel [26]. AVM tries to optimize each input variable in isolation. The chosen variable is randomly modified by increasing or decreasing a small amount, which is called an exploratory move. If the changes affect the fitness function, AVM applies a large amount in the same direction, which is called pattern moves. The pattern search is applied in the same direction as long as the fitness function is improved. The pattern search ends when it fails to optimize the fitness function. In this case, the search goes back to the exploratory moves to indicate a new direction on the same input variable. Once there are no further improvements of the input variable, the search moves to consider another variable, repeating the same process, until the branch is covered or no more variables can be improved.

3.4 *Memetic Algorithms*

Memetic algorithms (MA) combine both evolutionary algorithms and local search algorithms (e.g., a GA with a HC). In this case, MA implements a GA; additionally, at each generation, on each individual, a HC is applied to improve its fitness and reach a local optimum. MAs have been successfully applied to testing and showed better performance than evolutionary algorithms and local search algorithms in some cases [6, 8, 16].

4. RELATED WORK

In this section, we discuss the most closely related SBST approaches. In additions, the impact of the search space reduction on the performance of testing object oriented programs is explored.

4.1 *Search-Based Unit Testing*

Evolutionary Algorithms have already been applied to the problem of automated test data generation and have shown significant success. Tonella [36] applied GA to generate test cases for Java programs, and presented *eToc* tool for the Evolutionary Testing of Object Oriented (OO) software. In this approach, a population of individuals represents the test cases. New test cases were generated when a new branch is targeted. The fitness function is used to count the number of control dependences covered during test execution. One of the problems faced in separately tackling each branch, which is called the *structure-oriented approach* [28], is that when an uncovered branch is chosen as a target branch, the predicate of that branch might not be executed by any of the test cases in the population. In addition, no guidance is provided to the search on how to enter nested branches and cover them [28].

Several works addressed the issue of the *structure-oriented approach* and proposed fitness functions to guide the search process [3, 7, 27, 38]. Mainly, the fitness function combined two kinds of information: the *approach level* and the *branch distance*. The first is used to show how many of the conditional statements were not executed by a particular input to reach the target branch. The second computes the difference between a predicate value and a data input to execute the branch that leads to the target branch. The branch distance involves only numbers. As a result, if a predicate contains Boolean values, then it has only two different outcomes. This problem is called the flag problem [28]. In this case, several techniques were proposed for handling flag problems, for example testability transformations [21], and the chaining approach [12].

Arcuri and Yao [6] applied and analyzed different search algorithms on the testing of Java container classes. HC with random restarts, GA and MA were used and compared. Their empirical results showed that the MA results were the best among the algorithms. Moreover, a more advanced fitness function was proposed that maximize the number of branches and minimize the length of test cases.

EvoSuite [13] automatically generates and optimizes whole test suites towards satisfying a coverage criterion, e.g. branch coverage. EvoSuite uses GA that evolves and optimizes whole test suites to alleviate the problem that derive from infeasibility and difficulty of individual coverage goals. Recently, the GA search in the EvoSuite has been combined with local search (AVM) to optimize the values in a specific test case of a test suite [16]. Their result showed that the combined techniques increased the branch coverage by up 32% over GA.

Barsei et al. [8] proposed a hybrid global-local search (MA) tool for Java classes called *TestFul*. Their approach combines GA and HC, to generate tests that exercise the maximum number branches on the CUT. The former is used to search for the test that has higher coverage and reach all the interesting internal states of the CUT. The latter is used to target uncovered branches and analyze the controlled conditions of those branches to pick that ones are involved with numbers to cover. Our approach uses an algorithm that is derived from [6, 8] but which additionally incorporates a constant seeding strategy and uses method dependency relations (MDR).

4.2 *Search Space Reduction*

The goal of search-based algorithms for testing OO software is not only to generate test cases that instantiate the CUT followed by calling a sequence of method calls, but also to generate the desirable constructor parameters and the right method arguments. The large search space of distinct method numbers and parameter values can possibly hinder the search process. Thus, search space reduction deals with the elimination of the irrelevant methods and variable inputs from the input domain of the CUT, thereby reducing the size of the search space, which could potentially enhance the search process [31]. In spite of the large body of work on search-based software testing (SBST), there has been little investigation that addresses the relationship between search space and performance of search-based algorithms.

Harman et al. [20] were the first to empirically explore the search space reduction for the SBST. Their study analyzes the relationship between removing irrelevant input variables and SBST algorithms, including GA, HC and MA. In their work, static analysis was used to remove input variables that are irrelevant for determining whether a target branch will be executed or not, thereby reducing the search space. Their empirical study showed that irrelevant input removal improved the performance of the aforementioned SBST algorithms. However, the study focused on procedural programs and primitive parameters values. In a separate study, Binkley and Harman [9] conducted a simple experiment to show how the analysis of a predicate's dependence on parameters of a procedure can be used to reduce test data generation effort in evolutionary testing. Their initial results showed that the combination of analysis of predicate dependency with the optimized search required fewer fitness evaluations.

More recently, some researchers have addressed the issue of reducing the input domain of OO test data generation problems. Arcuri and Yao [6] proposed a technique called Dynamic Search Space Reduction (DSSR) that can be applied to any type of OO software. Their technique dynamically eliminates the read-only methods that do not change the state of the object from the search space. However, the study focused on a simple subset of Java programs, containers. As a result, a database for the common method names (e.g. insert, add, push) was used with string matching algorithms to determine whether a method is a read-only method or not. The empirical results showed that DSSR usage improved the efficiency of the search algorithms, particularly HC search, in terms of speed and number of steps to reach a global optimum, but applicability to non-containers was unclear. Some studies have suggested that containers have quite different behavior than more general code [19].

Barsei et al. [8] also proposed a semi-automated approach to augment the efficiency and speed-up the test generation with the *TestFul* tool. This is achieved by requiring the user to provide data regarding the effects of each method. A method can be: (1) a mutator, when it may change the object's state; (2) a worker, when it does not change the object's state but it may perform some computations, or (3) an observer, when it does not change the object's state and does not perform any additional computation. *TestFul* exploits the information and prunes methods from the test case that have no impact on the targeted branch before starting the HC search.

Sălcianu and Rinard [32] described a purity analysis technique for Java programs. Purity analysis is able to identify pure methods that have no side effects when executed, and can also recognize read-only and safe parameters even when the method is not pure. A parameter is read-only if the method does not mutate it and a safe parameter if it is read-only and the method does not produce any new externally visible heap paths to the objects reachable from these parameters.

EvoTest [33] and *eCrash* [31] approaches leverage purity analysis to reduce the input space of OO programming. The usage of the technique almost doubles the coverage/time performance of *EvoTest*. However, the user of the tool manually adds the "pure" annotation to complement the information generated automatically. On the other hand, the *eCrash* approach involves representing and evolving test cases using the Strongly-Typed Genetic Programming technique. The Extended Method Call Dependence Graph (EMCDG) is employed for constructing a method call sequence that puts the CUT into specific states. Then, parameter purity analysis is performed on the parameters of the method under test (MUT) and the purified EMCDG is obtained by removing the edges representing safe and read-only parameters from the EMCDG. Based on their empirical results, the inclusion of a parameter purity analysis phase into the process of test data generation has a significant improvement in the number of generation and computational time. Harman et al. [22] also proposed a domain reduction technique to exclude irrelevant parameters in the search space for aspect-oriented programs. They performed backward slicing to identify such irrelevant parameters [39]. The slice criterion is the predicate of a target aspectual branch, and the resulting program slice is used to exclude irrelevant parameters of target methods. Despite the fact that defined public fields were not considered in their domain reduction, their results showed a decrease in test effort with reduction, and an increase in the number of branches covered.

All approaches mentioned followed an approach similar to ours, but omit one or two pieces of information that are provided by our static analysis technique. Our static analysis provides information can be very helpful to reduce the search space and guide the search to create both values and sequences of method calls to exercise features that have impact on a target branch.

Regarding the reduction of the search space based on member class fields, we are aware of the work of Thummalapenta et al. [35]. In that work, the Seeker tool combines both dynamic symbolic execution (DSE) and static analysis. However, static analysis used in MAMDR differs completely from the static analysis used in their approach. Their approach uses method-call graphs while MAMDR uses Method Dependency Relations [42].

5. PROPOSED APPROACH

In this section, the concepts of our search-based approach are presented. Figure 2 illustrates MADMA's architecture.

The original source code is instrumented at bytecode level to measure the coverage values and calculate the fitness functions. In our experiments, we used Soot [40] for analyzing and instrumenting Java bytecode. The static analysis is used to identify method dependency relations based on the set of the fields that may be read or written by each method [42] and collect specific primitive values from predicates. The results are stored in a repository and used later by HC search. Then, GA is used to produce test cases that maximize the branch coverage of the CUT while minimizing the length of each test case. Finally, HC search attempts to cover every uncovered branch in the preceding search phase but exploits MDR to generate sequences of related method calls and initializes values using constants collected from the source code that would cover the target branches.

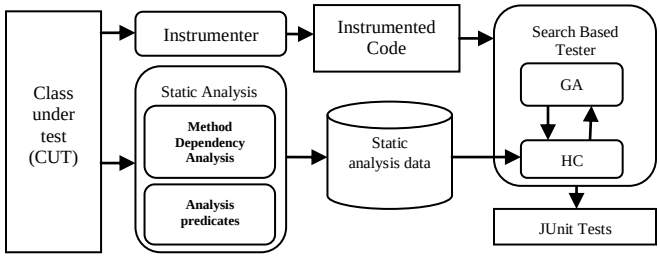

Figure 2: MADMR tool architecture

5.1 *Method Dependency Relations*

Zhang et al. [42] have introduced a systematic Method Dependence Relations (MDR) approach based on a hypothesis that two methods have dependence relations if the fields they read or write overlap. Their approach statically computes two types of dependence relations: *write-read* and *read-read*.

write-read relation: Given two methods f and g; f reads field x and g writes it, it is declared that f has *write-read* dependence relation on g.

read-read relation: If methods f and g both read the same field x, each method has a *read-read* dependence relation on the other.

More interestingly, their approach is able to define and merge the effects of the method calls: if a callee is a private method, it recursively merges its access field set into its callers. This helps reduce the search space size by only considering public methods that lead to executing targeted private methods.

In most cases, methods require instances of other classes to be used as arguments. To deal with that, we analyze the signatures of each public method and identify whether two methods have a possible dependence in terms of accessed data, (i.e. *accessed-data relation*) [41].

accessed-data relation: If a method f returns a non-primitive type and method g uses it as an a argument, it is declared that f has *accessed-data* dependence relation on g.

MDR is useful for testing two *write-read* related methods, as it has a high chance of exploring new program behaviors and states [42]. In addition, it is especially useful in the context of SBST, as the input domain of OO programs can be reduced by automatically identifying and eliminating *read-read* related methods that cannot give any further help from the search problem. In addition, MDR can also identify candidate methods that modify a specific member class field [6].

5.2 *Genetic Algorithm*

For an algorithm to be considered genetic, we need to define a representation of test cases as individuals, a fitness function, and the genetic operations.

A. Individual representation: An individual can be viewed as a sequence of functions calls. In this paper, we decided to use an individual representation similar to [8, 13], because it is easy to apply and manipulate. Each individual consists of a set of statements that are either a constructor or method call:

a. Constructor statement: represents a constructor call to generate a new instance of a selected class, e.g. **CDATAReader** CDATAReader_0= new **CDATAReader** (StdXMLReader_0).

b. Method statement: represents a public method call, e.g. CDATAReader_0.read(charArry_0,10,20). Parameters of constructors and method classes can be randomly generated and initialized depends on their types.

For a given CUT, the test cluster [37] is automatically defined. This is done by performing a static analysis of all the signatures of the public methods and constructors of the CUT, and adding each type encountered to the cluster. Returned non primitive objects are stored in a pool and served as a target object or parameter object for succeeding statement calls.

B. Fitness Function: GA uses fitness functions to determine if an individual is to be selected for reproducing in the subsequent generations. In this work, individual fitness is based on branch coverage, branch distance, and the length of the individual. In the GA search our goal is reaching the maximum number of covered branches while minimizing the length individuals [6, 8]. Thus, we use the fitness function in equation (1) to guide the GA search, and it is combined two objectives in a single function [6]:

$$f(i) = BR_{cov}(i) + (1 - \alpha)\ \frac{1}{1+len(i)} \text{ where } \alpha \in [0,1] \qquad (1)$$

While branch distance often gives good results, it can deceive the search and lead to longer individuals without increasing the coverage, which is called *bloat* [15]. As a result, we omit the branch distance in equation (1). This is because at the end of the search, we are only seeking to achieve high coverage with short length individuals [6].

C. Genetic Operations: in this work, common genetic operators are implemented:

a. Selection: In this operation two parents are selected for reproductions. We implement tournament selection [28]. In this selection mechanism, two individuals are randomly selected. Then, a random number r is generated $0 < r < 1$. Finally, we select the better individual if r is less than p ($p = 0.5$), otherwise the less fit individual is selected.

b. Crossover: this operator produces new individuals from the selected individuals. There are many different ways to implement crossover, such as single or multiple crossover points. We implement a single crossover point, where the two selected individuals are cut at a random single point.

c. Mutation: After crossover, the individuals are subjected to mutation. We randomly apply one of the following operations with probability $1/3$:

- **Remove:** A random number r is generated, where $1 \leq r \leq 5$, and r statements are removed from the individual at any random position in the individual.
- **Insert:** A random number r is generated, where $1 \leq r \leq 10$, and r statements are added to the individual at any random position in the individual. The input parameters for the statements are randomly generated
- **Change:** A random number r is generated, where $1 \leq r \leq 5$, and r statements have their parameters replaced with randomly selected values.

d. Elitism: At each new generation, the 10% of the population that have high fitness values are directly copied to the next new generation without any modification.

5.3 *Hill Climbing*

When the GA results stagnate, we employ hill climbing (HC) as the local search, similar to [6, 8]. For each branch uncovered in the GA, the individual that achieved best fitness for each reached branch is stored and used as a starting point for HC. Thus, the input of the HC is a list of all uncovered branches and the fittest individual for each branch. Every branch in the provided list is then processed in an attempt to cover it.

A branch is reached if its predicates are executed, while a branch is covered if its predicates are evaluated as true or false. An individual that reaches a branch is mutated and executed until the branch is covered or the stopping criterion is met, for example number of attempts. At each execution of the mutated individual, we keep track of new covered and reached branches and accordingly update the test suite.

A. Fitness Function: The fitness function that guides HC is similar to that which was used by Arcuri and Yao [6]. We apply the branch distance (BD) in the HC search because we target a single branch at a time and focus on the predicates of the target branch. Consequently, we use equation (2) for measuring BD.

$$BD(j) = \begin{cases} 0 & if\ the\ branch\ j\ is\ covered \\ \min(k) & if\ the\ branch\ j\ is\ reached \\ 1 & if\ the\ branch\ j\ is\ not\ reached \end{cases} \qquad (2)$$

Where k is a normalizing function, and we use the normalization function [3]: $k = x/(x+1)$, and x shows how far a

predicate is from obtaining its opposite value. For instance, for predicate $i < 10$ when the value of i is 2, then the distance to the false branch is $x = 10 - 2$ [13]. Finally, we integrate BD with the total branches coverage of the individual to guide the HC search in the following way:

$$f(i) = BR_{cov}(i) + (1 - BD(j)) \quad (3)$$

HC uses equation (3) as a fitness function to compare between the current and the mutated individual. However, if the two individuals have the same fitness, HC always picks the shortest individual [6].

Our approach explores the large space to generate candidate methods as well as specific constant values that help to cover target branches. Consequently, we analyze the targeted branch's predicates and precisely identify the type of elements that are involved in executing of the branch, e.g. member field, parameter method, or/and constant values. Then, we recommend methods and/or constant values for the following types of elements being involved in the condition's target branch:

a. Member field: To deal with class member fields, we followed a similar approach to the ones used by Thummalapenta et al. [35]. We precisely identify a member field and also leverage MDR to identify the related methods that write the targeted member field and help to achieve a desired value. If the target branch belongs to a non-public method, i.e. private, we also leverage MDR to identify all the public methods that call the targeted private method and recommend the identified related methods list to HC.

b. Parameter of method: We identify a parameter of method and also determine the type of the parameter, as well as the type of the method either public or private. Then, we also leverage MDR to identify the methods that call and/or have write-read relation with the targeted method. However, in some instances, it is impossible to reduce search space based on the parameters, because all parameters of a method can be involved in deciding whether a target branch is covered [20]. In spite of that, Harman et al. [20] showed that HC increases its search performance by removing irrelevant input variables:

c. Primitive Values: Rather than using random values, we apply a similar approach to that of Alshahwan et al. [1]. First, we collect constants from the target branch predicates. Then, we make a few changes to the constants and based on their types as inputs to the recommended related methods parameters these are used as input. Finally, with a certain probability we apply the following modifications based on the type of the constant:

- **Integer and Long:** We add/subtract a random number r to the constant value, with $-1 \leq r \leq 1$.
- **Float and Double:** We add/subtract a random number r to the constant value, with $-1 \leq r \leq 1$.
- **Boolean**: We only flip the value either true or false.
- **Character:** We randomly replace the value with another character.
- **Strings:** We apply one of three mutation operators as in [2]. (1) deletes the constant from the string value of the parameter methods in the individual. (2) inserts the constant in a random place into the string values of the parameter methods in the individual. (3) replaces the constant value with the parameter targeted method in the individual.

d. Array Values: We first leverage our static analysis information to determine the exact index i' for which an assignment helps to cover the target branch. Then, on the assignment of the index i', we generate input values depending on the component type of the array. We also use constant extracted from target branch predicates as input, rather than a random value.

Finally, we apply three different mutation operations to produce a modified version of the individual [6, 8]:

1- Insertion: Insert a random number r, where $1 \leq r \leq 10$, of methods that are randomly chosen from the identified related methods list in a random position in the individual.

2- Deletion: Remove a random number r, where $1 \leq r \leq 5$, of chosen methods from the identified related methods list from the individual, as well as remove all the methods that do not exist in the list.

3- Change: Change the parameters of a random number r, where $1 \leq r \leq 5$, of chosen methods or constructors in the individual with the modified constant.

Finally, the modified individual is then executed to see if the target branch is covered or if the fitness function is improved. In the former, the new individual is returned and is added to GA population and replaced with the least fit of an individual in the current population. In the latter, the fitter individual will be selected as a new starting point. HC repeats the aforementioned mutation operations until the attempt limit is reached. In this case, HC selects another uncovered branch, along with the individual that reaches the branch and tries to cover it.

6. EVALUATION

To validate our approach described in this paper, we compared its effectiveness against three different approaches: pure random testing, the EvoSuite [13] tool as a representative for search-based approaches, and a simple MA. EvoSuite is fully automatic and performs some code transformations to allow optimizations of string values. On the other hand, random testing (RT) has been recognized as an effective and fast testing technique, in which test cases consist of randomly selected methods with inputs randomly chosen from the input domain. Thus, to analyze the performance of random testing and MAMDR, we followed a random test generation strategy proposed by Ciupa et al. [11]. In addition, we compared a simple MA without method dependency relation (MDR) with our approach to show the effectiveness of our search space reduction approach in test data generation. MA uses both GA and HC [6, 8]. Unlike MAMDR, MA applies a simple HC to modify an individual. When HC targets an uncovered branch, it randomly performs one of the following actions: adding methods from the test cluster, removing statements, or changing the parameters of statements of the individual.

It would be very valuable to compare our approach performance with TestFul [8] and Seeker [35]. We could not use Seeker in our evaluation because it targets .Net programs, particularly C#, whereas MAMDR targets Java programs. In addition, TestFul is semi-automatic and it requires the user to provide some XML description of the CUT to enhance the efficiency of the approach. TestFul also requires the user to manually add additional classes which can be used as concrete implementations of the abstract classes and interfaces [8]. The large number of classes that we use in our experiments makes it harder to compare MAMDR with TestFul.

To evaluate MAMDR we consider several types of programs. We chose 4 open-source Java programs as used in the EvoSuite experiments [16]. We also included DateTimeFormat and Fraction classes where search-based approaches, like EvoSuite, did not achieve high coverage. However, not all classes contain numeric or string constants nor predicates, which are easy to analyze.

Therefore, this set of subjects contains three container classes, which are taken from the work of Sharma et al. [34], to see whether our approach has a negative effect on the performance of the search process when its power is not needed. Table 1 lists our evaluation subjects, including their number of public classes, lines of code[2], and number of instrumented branches.

Case Study		# Classes	LOC	#Branches
Commons CLI	[8]	11	667	288
Commons Codec	[8]	26	2650	1371
NanoXML	[8]	12	1532	591
org.jdom2	[8]	20	2869	1108
org.joda.time.format.DateTimeFormat	[8]	1	365	145
Fraction	[4]	1	252	140
StringTokenizer	[28]	1	122	72
AvlTree	[29]	1	306	148
BinomialHeap	[29]	1	185	62
TreeMap	[29]	1	481	158

Table 1: Case Study Subjects

6.1 *Research Questions*

Having defined the case study subjects, we now address the following research questions:

RQ1: Does MAMDR achieve higher branch coverage than representative test generation tools?

To answer this question, we ran RT, EvoSuite, MA, and MAMDR on each target subject with a time limit. The original source code of each subject was instrumented to measure the branch coverage of each approach.

RQ2: What is the impact of using constants from target branches predicates for seeding?

For this question, we first ran two different versions of MAMDR, one version seeds the search process with the constant values (denoted as **MWS**), and the other version without seeding (denoted as **MNS**).

6.2 *Evaluation Setup*

We next describe our evaluation setup in order to answer the preceding two research questions. Search algorithms have many parameters to adjust; in this experiment we followed similar settings in [6]. The GA uses the fitness function defined in equation (1), with $\alpha = 0.5$. The GA also uses a single point crossover with probability 0.8. Mutation probability of an individual is 0.9. The population size is 100, and the length of the individual is set to 80. Tournament with size 2 is used in the selection phase. The elitism is set to 10% of the population size. HC uses the fitness function defined in equation (3). We apply HC after five consecutive generations without any further improvements in the total branch coverage, i.e. the population of the search had stagnated. The number of attempts for each target uncovered branch is set to 1,000 which means each uncovered branch gets at least 1,000 fitness evaluations whenever being selected. We also considered the constant seeding from the branch predicates with the probability 0.8.

We ran EvoSuite with default configurations, and only tuned the running time for test generation to the required time limit. The length of test cases in the random testing is set to 200 [18]. The probability of creating a new instance of a chosen class rather than using existing ones = 0.25. However, with probability 0.1, the instance of the chosen class is set to null. For string values, characters are chosen randomly from the set of 95 printable ASCII characters (0x20–0x7E) [25]. All the experiments were conducted on a machine with Intel Core 2 Quad CPU @ 2.66 GHz and 8 GB RAM.

[2] http://Javancss.codehaus.org/

To evaluate the statistical difference of our approach, we followed the guidelines in [4]. For each approach, we set the time limit 5 minutes, and run 30 times for different random seeds on each test class (not per test subject program).

7. Results

This section provides a summary of the results with respect to the research questions.

7.1 *Coverage Results*

Table 2 summaries the result obtained by the experiment for all the test cases subjects. The table shows the average of the branch coverage value over the 30 runs with different random seeds. We highlighted in bold where the highest branch coverage is achieved by each approach with statistical significance, respectively. The statically difference has been calculated with Mann-Whitney U at the 95% confidence level.

Test subject	RT(%)	EvoSuite(%)	MA(%)	MNS (%)	MWS (%)
Commons CLI	96.88	95.67	96.83	96.84	**99.28**
Commons Codec	91.59	89.34	91.94	**92.33**	**93.20**
NanoXML	63.32	59.20	65.67	**70.85**	**73.68**
org.jdom2	82.50	80.19	80.11	**82.40**	**86.39**
DateTimeFormat	82.09	68.69	81.15	**83.75**	**89.06**
Fraction	**93.52**	85.45	90.36	**93.10**	92.93
StringTokenizer	62.50	**63.89**	62.5	62.50	**86.62**
AvlTree	95.27	70.50	95.27	95.27	95.27
BinomialHeap	90.32	88.71	**93.55**	**93.55**	**93.55**
TreeMap	82.91	82.91	82.91	82.91	82.91
Average:	**84.09**	**78.46**	**84.03**	**85.35**	**89.93**

Table 2: Average branch coverage Achieved by RT, EvoSuite, MA, MNS, and MWS.

7.1.1 Comparison with RT

The results in table 2 show that **MNS** outperforms RT on 3 test subjects in the branch coverage. Coverage levels were identical between **MNS** and RT for 4 test subjects, particularly container classes. NanoXML shows the highest improvement with a 7.53% increase in coverage. The reason why RT achieves a lower branch coverage than **MNS** can be explained by the fact that some constructors of classes in the NanoXML require instances of other classes and/or specific values used as arguments. For example, the constructor of class `StdXMLReader` requires a `Reader` object (the input for the XML data), and string values as arguments. RT, thus, creates many invalid objects of `StdXMLReader` due to the large size and complexity of the search space, and then fails to reach desirable states that help to cover target branches. On the other hand, static analysis used in **MNS** helps to identify related methods that lead to cover branches. For instance, **MNS** identifies the `stringReader` method, which only takes one string argument, and returns a valid `StdXMLReader` object instance, thereby reducing the search space size. Invoking `stringReader` allows **MNS** to create many valid `StdXMLReader` objects that can be used to reach many desirable states that help to cover branches.

We also noticed that **MNS** showed no substantial branch coverage improvement over RT in Common Codec and Commons CLI, where RT previously observed to be very effective in testing Apache Commons programs [42]. One main reason is that Common Codec has few path constraints and its methods can be called without any specific order to initialize objects, suggesting **MNS** strength lies in testing classes that require complex input sequences. Moreover, RT also did a little better than **MNS** for the Fraction class. This can be explained by considering that the Fraction class is immutable, which means constructors of the class updates its member fields, and if parameters of the constructors are not valid an exception is thrown, which hinders the search process [8]. In our experiments, the length of RT test cases was set to 200.

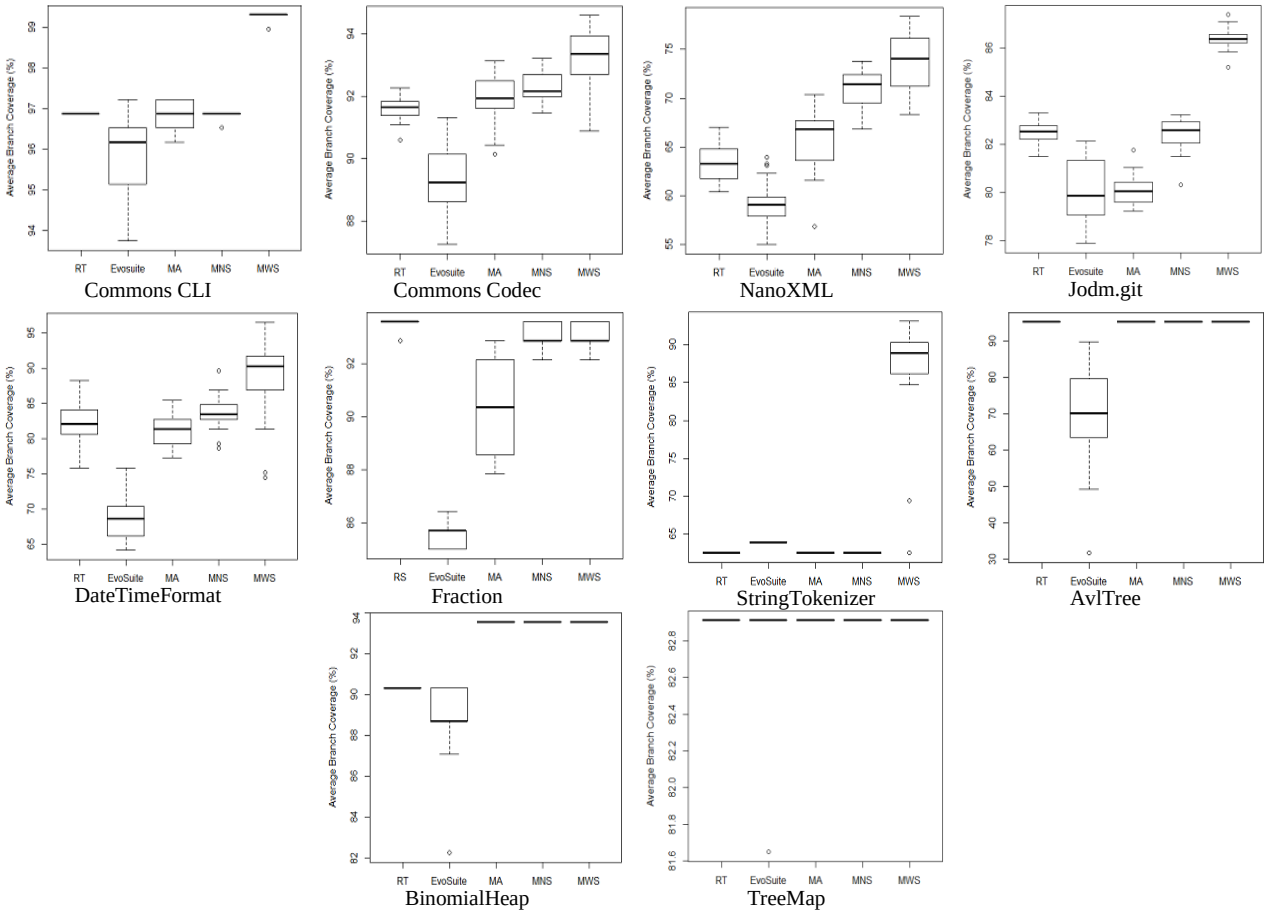

Figure 3: Average Branch Coverage of each of the 5 approaches on each test subject

That allows RT to randomly create many desirable object instances in each test case. The capability to generate a number of valid object instances helps RT to cover many branches of the Fraction class. However, the static analysis used in **MNS** helps to identify the constructor of the class is responsible of writing its fields. This helps **MNS** to concentrate on creating a valid Fraction object instance and avoiding throwing exceptions, and thus the search process is improved [8].

7.1.2 Comparison with EvoSuite

Our results show that **MNS** achieved higher branch coverage than EvoSuite for all subjects. Although EvoSuite creates method call sequences with the assistance of transformed String methods like String.equals to calculate distance measurements to the branches [13], it fails to generate method call sequences that cover very difficult branches. We identified two possible reasons for the lowest branch coverage of EvoSuite. First, the measurement in the branch distance offers little guidance to explore a large search space and find input data to cover difficult branches. Second, when an individual in EvoSuite is a set of test cases, each of which consists of a sequence of method calls, then the size of the search space is very large [16]. As a result, it is difficult for EvoSuite to mutate a primitive value and find a desirable value input to cover a target branch, since the probability of it being mutated during the search is very low [16]. Thus, EvoSuite finds it hard to make progress towards the optimal solution by only using mutation and crossover operations. The aforementioned two reasons for branches to remain uncover are all related to the size of the search space, a weakness of EvoSuite observed in recent approaches [16, 17, 30]. Indeed, the exclusion of irrelevant methods from the search space can effectively improve the performance of EvoSuite because the mutation operator will be concentrating its effort on methods that can influence coverage of a target branch [20].

7.1.3 Comparison with MA

Table 2 also shows the comparison results on branch coverage achieved by MA and MNS. We can observe that MNS outperforms MA in 5 out of 10 test subjects. In the remaining five test subjects, MA and MNS achieve exactly the same coverage. Among these five subjects where MA and MNS achieve the same branch coverage is Commons CLI. Commons CLI has only a few constraints that need to be satisfied [42]. Therefore, the majority of Commons CLI branches are trivial and randomly picking methods and finding their arguments across the whole search space can achieve good results. In summary, input domain search space reduction can cause an increase in branch coverage, particularly, for programs that contain branches requiring specific method calls ordering or arguments.

7.1.4 Impact of seeding constants

As might be expected, seeding improves branch coverage 7 of 10 test subjects (Table 2). Branch coverage was identical for container classes. Noticeable improvements were obtained for Commons Codec, Commons CLI, NanoXML, org.jdom2, and StringTokenizer. The highest improvement with 5.31% was recorded in DateTimeFormat test subject. In DateTimeFormat class, most of the branch targets are contained in private methods, and depended on a string values. Approaches like EvoSuite or RT

might need to run for very long time to cover these branches. MWS can cover these target branches much quicker for two possible reasons. First, collecting constants from predicates of the target branches helps **MWS** to seed these constants into the search process, and cover branches that depend on these specific constants. Second, identifying related methods helps **MWS** to generate sequence of method calls to a target branch with desired values for member fields and method arguments.

Figure 3 shows a box-plot of the actual average branch coverage achieved over 30 runs of each approach on each test subject. As the figure shows, in many test subjects **MWS** achieves higher branch coverage than other approaches. For Commons CLI, Commons Codec, and StringTokenizer **MWS** shows the highest coverage. In each case, **MWS** seeded valid constant string values to the tested methods to cover specific branches which guided the search towards additional nested branches. However, these values are difficult to generate due to the randomized generation in each other approach. We also notice that **MWS** shows identical coverage over 30 runs compared to **MNS** for Fraction class. The primary reason is that this class is a number implementation and has methods that accept numbers, which contain few constant-using predicates. As a result, both approaches relied on fitness function to guide the search to generate input data that cover target branches.

Despite **MWS** improving branch coverage on most test subjects, it still does not achieve 100% branch coverage. The simple explanation is that some classes might contain branches that are included in private methods which are not called by any public methods [16]. In addition, some branches required complex data inputs to be covered. For example, some methods in the `StdXMLReader` class, which in the NanoXML test subject, require a file containing XML data as input. These types of inputs are difficult to generate, and thus MWS generates ineffective tests.

What came as a surprise is that RT outperforms EvoSuite in most test subjects. One explanation would be that the length of the test case for RT is 200 [18], which is three times as long as EvoSuite's. Although it may be possible to find parameter settings for which EvoSuite performs better, discovering parameter settings can be considered computationally expensive [5]. For this reason, we postpone finding better parameters to future work. We will consider adopting different parameters settings across all different approaches; in particular, we are concerned to adopt the same settings on all representative approaches, such as the same length of the test case, when the defaults of tools may not be best.

8. Threats to Validity

In this section we discuss the main threats that could affect the validity of our results.

8.1 *Internal Validity:*

The major internal threat that could affect our results is the probability of having faults in our instrumentation. To minimize this threat, we carefully tested our instrumentation framework and manually tested instrumented source code for several program subjects. Another potential threat to internal validity could be with randomized algorithms. Therefore, we ran our experiments for 30 times and applied rigorous statistical procedures.

8.2 *External Validity:*

The external validity is how generalizable our results are based on our selection of test subjects. The test subjects in this experiment were different types of programs and their size varied by an order of magnitude. We included open source projects and container classes. In addition, the selection test subjects have been widely used in other empirical studies in SBST.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed MAMDR, a fully automatic tool that utilizes three different approaches: genetic algorithms, hill climbing, and method dependence relations to achieve high code coverage. To evaluate MAMDR, we conducted evaluations on several open source programs and container classes. Our results showed that MAMDR demonstrated significant improvements in branch coverage compared to purely random testing and the search-based EvoSuite.

With our approach, related methods, which are based on their fields, are exploited to modify particular fields or arguments in order to cover a branch that is required for a certain execution path. This is particularly useful to handle a large search space and to generate sequences of method calls for classes with complicated constraints branches.

The individual presented concepts of our automated search-based test generation have (in many cases) been applied in other approaches to generate test cases for OO programs, like Java, but the combination of methods and exploiting of all information available is key to overall success. We showed that it can be difficult for search-based approaches to generate test cases that include good method sequences and arguments, due to the application of a pure randomized algorithm in the mutation phase. We also showed how our novel seeding approach exploited method dependence relations to increase the effectiveness of the SBST.

In future work, we will focus on integrating method dependence relations into the genetic algorithm phase and enhancing its mutation operators. Further, we also plan to capture object instances from different search phases and then exploit these object instances to guide the search in generating test cases. Finally, we plan to conduct further experiments and analyses on MAMDR's coverage and efficiency when these ideas are implemented.

ACKNOWLEDGMENT

The authors would like to thank Tripoli University, Tripoli, Libya for its support to Ali Aburas.

10. REFERENCES

[1] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in IEEE/ACM Int. Conference on Automated Software Engineering (ASE), 2011, pp. 3–12.

[2] M. Alshraideh, L. Bottaci, "Search-based software test data generation for string data using program-specific search operators," Software Testing Verification and Reliability, 16(3), 2006, pp. 175-203.

[3] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," Software Testing, Verification and Reliability (STVR), 2011.

[4] A. Arcuri and L. Briand "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in ACM/IEEE International Conference on Software Engineering (ICSE), 2011, pp. 1–10.

[5] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," Empirical Software Engineering, February 2013.

[6] A. Arcuri and X. Yao, "A memetic algorithm for test data generation of object-oriented software," in Proceedings of the IEEE Congress on Evolutionary Computation, 2007.

[7] A. Baresel , H. Sthamer and M. Schmidt "Fitness Function Design to Improve Evolutionary Structural Testing", Proc. Genetic and Evolutionary Computation Conf., pp.1329 -1336 2002

[8] L. Baresi, P. L. Lanzi, and M. Miraz, “Testful: an evolutionary test approach for Java,” in ICST'10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation. IEEE Computer Society, 2010, pp. 185-194.
[9] D. Binkley and M. Harman, “Analysis and Visualization of Predicate Dependence on Formal Parameters and Global Variables,” IEEE Transactions on Software Engineering, v.30 n.11, p.715-735, November 2004
[10] C. Cadar, D. Dunbar, and D. R. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” In OSDI, 2008.
[11] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, “Experimental assessment of random testing for object-oriented software,” In ISSTA ’07: Proceedings of the 2007 International symposium on Software testing and analysis, pages 84–94, New York, NY, USA, 2007. ACM
[12] R. Ferguson and B. Korel, “The chaining approach for software test data generation,” ACM Trans. Softw. Eng. Methodol., vol. 5, no. 1, pp. 63–86, 1996.
[13] G. Fraser and A. Arcuri, “Whole test suite generation,” in International Conference on Software Quality (QSIC 2011), 2011.
[14] G. Fraser and A. Arcuri, “The seed is strong: Seeding strategies in search-based software testing,” in IEEE Int. Conference on Software Testing, Verification and Validation (ICST), pages 121–130, 2012.
[15] G. Fraser and A. Arcuri, “Handling test length bloat,” Software Testing, Verification and Reliability, 2013.
[16] G. Fraser, A. Arcuri, and P. McMinn, “Test suite generation with Memetic algorithms,” In GECCO, 2013.
[17] J. P. Galeotti, G. Fraser, and A. Arcuri, “Improving Search-based Test Suite Generation with Dynamic Symbolic Execution,” in IEEE International Symposium on Software Reliability Engineering (ISSRE), 2013.
[18] A. Groce, “Coverage rewarded: Test input generation via adaptation-based programming,” In International Conference on Automated Software Engineering, pages 380–383, 2011.
[19] A. Groce, C. Zhang, M. A. Alipour, E. Eide, Y. Chen, J. Regehr, "Help, help, i'm being suppressed! The significance of suppressors in software testing," issre, pp.390-399, 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 2013
[20] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener, “The Impact of Input Domain Reduction on Search-Based Test Data Generation,” in ESEC/SIGSOFT FSE, 2007, pp. 93-101.
[21] M. Harman , L. Hu , R. Hierons , J. Wegener , H. Sthamer , A. Baresel and M. Roper “Testability Transformation”, IEEE Trans. Software Eng., vol. 30, no. 1, pp.3 -16 2004
[22] M. Harman, F. Islam, T. Xie, and S. Wappler, "Automated test data generation for aspect-oriented programs," in AOSD, 2009.
[23] K. Inkumsah and T. Xie, “Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execute on,” in Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE’08). Washington, DC, USA: IEEE Computer Society, 2008, pp. 297–306.
[24] P. McMinn, M. Holcombe, “Hybridizing evolutionary testing with the chaining approach,” in: Genetic and Evolutionary Computation Conference (GECCO), 2004, pp. 1363–1374.
[25] 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), 2012.
[26] B. Korel. “Automated software test data generation,” IEEE Transactions on Software Engineering, pages 870–879, 1990.
[27] G. McGraw, C. Michael, and M. Schatz. “Generating software test data by evolution,” IEEE Transactions on Software Engineering, 27(12):1085-1110,2001.
[28] P. McMinn, “Search-based software test data generation: A survey. Software Testing, Verification and Reliability,” 14(2):105-156, 2004.
[29] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in ICSE, 2007.
[30] Y. Pavlov and G. Fraser, “Semi-automatic Search-Based Test Generation,” in 5th International Workshop on Search-Based Software Testing (SBST’12) at ICST’12, 2012, pp. 777-784.
[31] J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. de Vega, “Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software,” Information and Software Technology, 2009. (In Press).
[32] A. Sălcianu and M. Rinard, “A combined pointer and purity analysis for Java programs,” Technical Report MIT-CSAIL-TR-949, MIT, May 2004.
[33] A. Seesing and H.G. Gross, “A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software,” International Transactions on Systems Science and Applications, Vol.1, No.2, pp.127-134, 2006.
[34] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, “Testing container classes: Random or systematic?” in Fundamental Approaches to Software Engineering, 2011.
[35] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. “Synthesizing method sequences for high-coverage testing,” In OOPSLA, pages 189-206, 2011.
[36] P. Tonella, “Evolutionary testing of classes,” In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pages 119-128, 2004.
[37] S. Wappler and F. Lammermann, “Using evolutionary algorithms for the unit testing of object-oriented software,” in GECCO’05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation. ACM, 2005, pp. 1053–1060.
[38] J. Wegener, A. Baresel, and H. Sthamer. “Evolutionary test environment for automatic structural testing,” Information and Software Technology, 43(14):841-854, 2001.
[39] M. Weiser, "Program Slicing," IEEE Trans. Software Eng., vol. 10, no. 4, pp. 352-357, 1984.
[40] R. Vall´ee-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, “Soot - a Java bytecode optimization framework,” in CASCON, 1999, pp. 125–135.
[41] S. Zhang, Y. Bu, X. Wang, and M. D. Ernst. “Dependence-guided random test generation. CSE 503 Course Project Report,” University of Washington. URL: www.cs.washington.edu/homes/szhang/gencc.pdf, May 2010.
[42] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, “Combined static and dynamic automated test generation,” in ISSTA, 2011.