Search-Based Software Testing: Past, Present and Future

Phil McMinn
University of Sheffield, Department of Computer Science
Regent Court, 211 Portobello, Sheffield, S1 4DP, UK

***Abstract*—Search-Based Software Testing is the use of a meta-heuristic optimizing search technique, such as a Genetic Algorithm, to automate or partially automate a testing task; for example the automatic generation of test data. Key to the optimization process is a problem-specific fitness function. The role of the fitness function is to guide the search to good solutions from a potentially infinite search space, within a practical time limit.**

Work on Search-Based Software Testing dates back to 1976, with interest in the area beginning to gather pace in the 1990s. More recently there has been an explosion of the amount of work. This paper reviews past work and the current state of the art, and discusses potential future research areas and open problems that remain in the field.

I. Introduction

The first publication on what has become known as 'Search-Based Software Testing' appeared in 1976, and was the work of two American researchers, Webb Miller and David Spooner [1]. Their approach was a simple technique for generating test data consisting of floating-point inputs, and was a completely different approach to the test data generation techniques being developed at the time, which were based on symbolic execution and constraint solving. In Miller and Spooner's approach, test data were sought by executing a version of the software under test, with these executions being guided toward to the required test data through the use of a 'cost function' (hereon referred to a fitness function), coupled with a simple optimization process. Inputs that were 'closer' to executing a desired path through the program were rewarded with lower cost values, whilst inputs with higher cost values were discarded.

Miller and Spooner did not continue their work in test data generation[1], and it was not until 1990 that their research directions were continued by Korel [3], [4]. In 1992, Xanthakis applied Genetic Algorithms to the problem [5]. Since then there has been an explosion of work, applying meta-heuristics more widely than just test data generation. Search-based optimisation has been used as an enabler to a plethora of testing problems, including functional testing [6], [7], temporal testing [8], [9], [10], integration testing [11], regression testing [12], stress testing [13], mutation testing [14], test prioritisation [15], [16], interaction testing [17], state machine testing [18] and exception testing [19]. The explosion of interest can be seen in Figure 1, a bar chart of the number of publications in Search-Based Software Testing in each year since 1976 where the authors applied search-based optimisation to a problem in testing. Search-based approaches have also been applied to problems in the wider area of software engineering, leading Harman and Jones to coin the phrase 'Search-Based Software Engineering' [20] in 2001. The term 'Search-Based Software Testing' began to be used to refer to a software testing approach that used a metaheuristic algorithm, with the amount of work in Search-Based Test Data Generation alone reaching a level that led to a survey of the field by McMinn in 2004 [21].

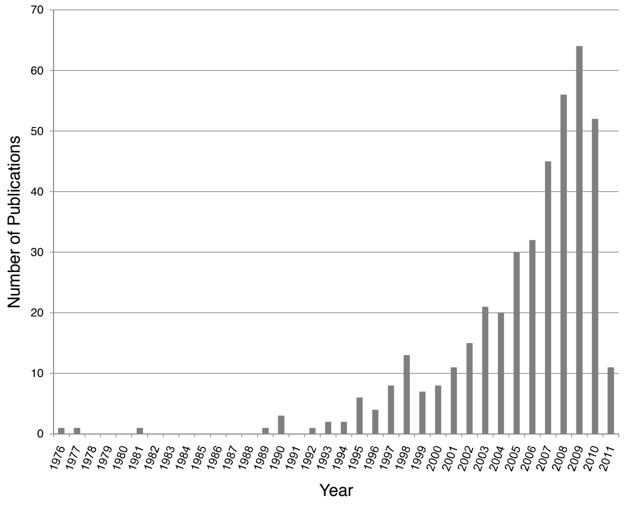


Figure 1. Publications in Search-Based Software Testing, dating back to 1976. The figures for 2010 and 2011 are based on partially complete data at the time of writing

This paper provides a brief introduction to the ideas and behind Search-Based Software Testing, including some examples of past work. The rest of the paper is devoted to open problems that still exist in the area and may form the basis of future work. Section II gives an overview of the main optimization algorithms that have been applied in Search-Based Testing, including Hill Climbing, Simulated Annealing and Genetic Algorithms. Key to any optimisation approach is the definition of a fitness function - it is the guidance provided by the fitness function that allows the search to find good solutions in a reasonable time frame. Section III details some example fitness functions used in three example applications of Search-Based Testing: temporal testing, functional testing and structural testing.

[1]Webb Miller has since gone on to forge a highly-successful career in computational biology, including award-winning work on algorithms for analysing DNA sequences. His research on sequencing the wooly mammoth genome led to his listing in the 2009 Time 100 [2].

Section IV discusses future directions for Search-Based Software Testing, comprising issues involving execution environments, testability, automated oracles, reduction of human oracle cost and multi-objective optimisation. Finally, Section V concludes with closing remarks.

II. Search-Based Optimization Algorithms

The simplest form of an optimization algorithm, and the easiest to implement, is random search. In test data generation, inputs are generated at random until the goal of the test (for example, the coverage of a particular program statement or branch) is fulfilled. Random search is very poor at finding solutions when those solutions occupy a very small part of the overall search space. Such a situation is depicted in Figure 2, where the number of inputs covering a particular structural target are very few in number compared to the size of the input domain. Test data may be found faster and more reliably if the search is given some guidance. For meta-heurstic searches, this guidance can be provided in the form of a problem-specific *fitness function*, which scores different points in the search space with respect to their 'goodness' or their suitability for solving the problem at hand. An example fitness function is plotted in Figure 3, showing how - in general - inputs closer to the required test data that execute the structure of interest are rewarded with higher fitness values than those that are further away. A plot of a fitness function such as this is referred to as the *fitness landscape*. Such fitness information can be utilized by optimization algorithms, such as a simple algorithm called Hill Climbing. Hill Climbing starts at a random point in the search space. Points in the search space neighbouring the current point are evaluated for fitness. If a better candidate solution is found, Hill Climbing moves to that new point, and evaluates the neighbourhood of that candidate solution. This step is repeated, until the neighbourhood of the current point in the search space offers no better candidate solutions; a so-called 'local optima'. If the local optimum is not the global optimum (as in Figure 3a), the search may benefit from being 'restarted' and performing a climb from a new initial position in the landscape (Figure 3b).

An alternative to simple Hill Climbing is Simulated Annealing [22]. Search by Simulated Annealing is similar to Hill Climbing, except movement around the search space is less restricted. Moves may be made to points of lower fitness in the search space, with the aim of escaping local optima. This is dictated by a probability value that is dependent on a parameter called the 'temperature', which decreases in value as the search progresses (Figure 4). The lower the temperature, the less likely the chances of moving to a poorer position in the search space, until 'freezing point' is reached, from which point the algorithm behaves identically to Hill Climbing. Simulated Annealing is named so because it was inspired by the physical process of annealing in materials.

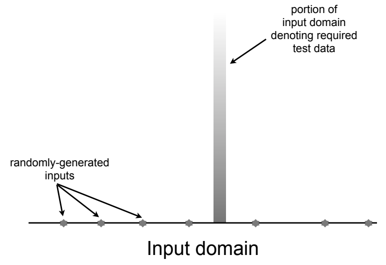
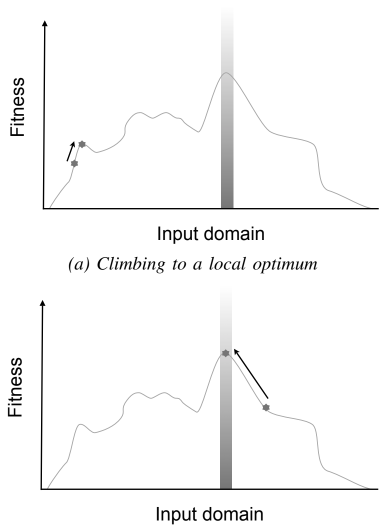

portion of input domain denoting required test data
randomly-generated inputs
Input domain

Figure 2. Random search may fail to fulfil low-probability test goals



(a) Climbing to a local optimum

(b) Restarting, on this occasion resulting in a climb to the global optimum

Figure 3. The provision of fitness information to guide the search with Hill Climbing. From a random starting point, the algorithm follows the curve of the fitness landscape until a local optimum is found. The final position may not represent the global optimum (part (a)), and restarts may be required (part (b))

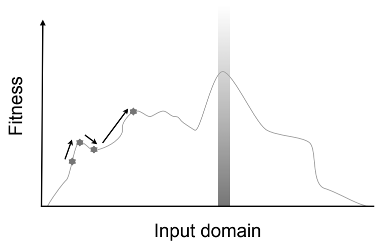


Figure 4. Simulated Annealing may temporarily move to points of poorer fitness in the search space

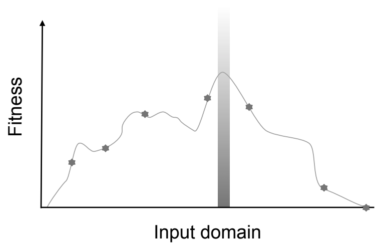


Figure 5. Genetic Algorithms are global searches, sampling many points in the fitness landscape at once

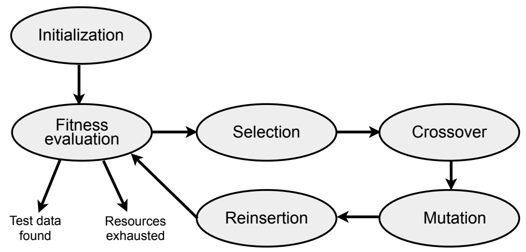


Figure 6. Overview of the main steps of a Genetic Algorithm

Hill Climbing and Simulated Annealing are described as 'local' search approaches, because they consider only one solution at a time, and make moves only in the local neighbourhood of those solutions. Genetic Algorithms, on the other hand, are a form of 'global' search, sampling many points in the search space at once, as shown in Figure 5. Genetic Algorithms are inspired by Darwinian evolution and the concept of survival of the fittest. Each point in the search space currently under consideration is referred to as an 'individual' or a 'chromosome'. The current set of individuals currently under consideration are collectively referred to as the current 'population'. The main loop of a Genetic Algorithm can be seen in Figure 6. The first population is randomly generated, and each individual is evaluated for fitness. A selection mechanism, biased towards the best individuals, decides which individuals should be parents for crossover. During crossover, elements of each individual are recombined to form two offspring individuals that embody characteristics of their parents. For example, two strings '`XXX`' and '`OOO`' may be spliced at position 2 to form two children '`XOO`' and '`OXX`'. Subsequently, elements of the newly-created chromosomes are mutated at random, with the aim of diversifying the search into new areas of the search space. This may, for example, involve overwriting one of the characters of the above strings with a new character. For problems involving real values, mutation may instead involve incrementing or decrementing values pertaining to one of the elements of the chromosome. Finally, the next generation of the population is chosen in the 'reinsertion' phase, and the new individuals are evaluated for fitness. This cycle continues, until the Genetic Algorithm finds a solution or the resources allocated to the search (*e.g.* a time limit or a certain number of fitness evaluations) are exhausted. For an excellent introduction on getting starting with Genetic Algorithms in Search-Based Software Engineering, see Whitley's tutorial papers [23], [24].

In general, there are two requirements that need to be fulfilled in order to apply a search-based optimization technique to a testing problem [20], [25]:

1) **Representation.** The candidate solutions for the problem at hand must be capable of being encoded so that they can be manipulated by the search algorithm - usually as sequences of elements as for chromosomes with a Genetic Algorithm.
2) **Fitness function.** The fitness function guides the search to promising areas of the search space by evaluating candidate solutions. The fitness function is problem-specific, and needs to be defined for a new problem. The next section discusses some fitness functions that have been used by different authors in Search-Based Software Testing.

Harman [25] argues that software engineers typically already have a suitable representation of their problem. This is automatically the case for test data generation, where the input vector or sequences of inputs to the software under test can be optimized more or less directly. Harman and Clark [26] further argue that many software engineers naturally work with software metrics that can form the basis of fitness functions, leaving only the application of an optimization technique the only step left in order to apply a search-based approach. The next section serves to demonstrate some of the areas to which search-based optimization has been applied in testing, and the fitness functions used in each case.

III. Example Application Areas and Fitness Functions

Temporal Testing

Temporal testing involves probing a component of a system to find its best-case and worst-case execution times (BCET and WCET, respectively). Often these can be approximated by static analysis, but these approximations are often conservative over-approximations in the case of WCET, and under-approximations in the case of BCET. Only actual execution of the software can reveal concrete times. BCET and WCET are of paramount importance in safety-critical systems. An example is that of an air-bag controller, which must monitor the deceleration profile of a car and decide when to launch the air bag. If the air bag is released too early, the action may be premature, while releasing the air bag too late may prove fatal to the driver.

Search-Based Software Testing has been found to be an effective means of finding inputs to a piece of software that result in long or short execution times [8], [9], [10]. The fitness function is simply the execution time of the software, found by simply running it with an input. For BCET, the search attempts to minimise the fitness function, in order to find shorter execution times; whilst for WCET, the search attempts to maximise the fitness function, in order to reveal longer execution times. Conversely to static analysis, the times revealed by the search tend to under-approximate WCET and over-approximate BCET in practice. However, the timings found can be used in conjunction with those derived through static analysis to give an interval in which the actual times must lie [27].

Functional Testing

A famous example of search-based functional testing is the testing of the car parking controller of DaimlerChrysler [6], [7]. The parking controller is responsible for identifying a suitable parking space, and then automatically manoeuvring the car into the space without colliding with any other objects. The controller was tested in simulation using Search-Based Software Testing. The fitness function used was simply the shortest distance to a point of collision during the 'park'. The search would then attempt to minimize the distance in order to reveal situations in which the controller was faulty; *i.e.* led to a possible collision. The evolutionary search used generated parking scenarios, which were then simulated by the controller, and the closest distance to a collision recorded through the simulation to give a fitness value. Faults were revealed by the search pertaining to an initial version of the system, whereby the car began from a position that was very close to another object situated to its side.

Structural Testing

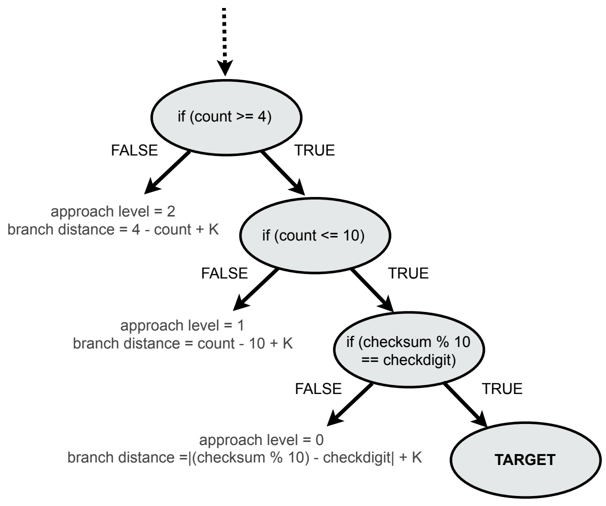
The original use of search-based techniques in testing was to generate test data for the execution of paths, in the Miller and Spooner approach [1]. Structural testing has been the application area that has attracted the most attention in Search-Based Software Testing. Work has been undertaken that has developed fitness functions for path coverage, branch coverage and data flow coverage amongst others [28]. Typically, the program under test is instrumented, and the program executed with inputs suggested by the search. The path taken through the program is then compared with some structure of interest for which coverage is sought. The C function of Figure 7 is a typical routine for which coverage might be sought using search-based techniques. The function checks a registry number assigned by the Chemical Abstracts Services to uniquely identify every chemical described in the scientific literature, and involves a checksum calculation.

A popular fitness function used for finding test data to cover individual branches was proposed by Wegener *et al.* [28], and incorporates two metrics, known as the *approach level* and the *branch distance*. The approach level is the number of the target's control dependent nodes that were not executed by the path for a given input. It is equivalent the number of levels of nesting left unpenetrated by the path en route to the target for structured programs. Suppose a valid CAS number string is required for the execution of the true branch from line 19. For this target, the approach level is 2 if no invalid characters are found in the string (*i.e.* characters that are not digits or hyphens), but there are too few digits, leading to the false branch is taken at line 17. If instead the string has too many valid digits, the true branch is taken at node 17, but the target is then missed because the false branch was taken at node 18. In this instance, the approach level is 1. When the checksum calculation is reached at line 19, the approach level is zero.

When the execution of a test case diverges from the target branch at some approach level, the *branch distance* is computed. The branch distance is a measure of 'how close' an input came to satisfying the condition of the predicate at which control flow went 'wrong'; *i.e.*, how near the input was to executing the required branch and descending to the next approach level. For example, suppose execution takes the false branch at node 17 in Figure 7, but it is the true branch that needs to be executed. Here, the branch distance is computed using the formula $4 - count + K$. K

```
(1)    int cas_check(char* cas) {
(2)      int count = 0, checksum = 0, checkdigit = 0, pos;
(3)
(4)      for (pos=strlen(cas)-1; pos >= 0; pos--) {
(5)        int digit = cas[pos] - '0';
(6)
(7)        if (digit >= 0 && digit <= 9) {
(8)          if (count == 0)
(9)            checkdigit = digit;
(10)         if (count > 0)
(11)           checksum += count * digit;
(12)
(13)         count ++;
(14)       }
(15)     }
(16)
(17)     if (count >= 4)
(18)       if (count <= 10)
(19)         if (checksum % 10 == checkdigit)
(20)           return 0;
(21)         else return 1;
(22)       else return 2;
(23)     else return 3;
(24)   }
```

(a) Code


if (count >= 4)
FALSE
TRUE
approach level = 2
branch distance = 4 - count + K
if (count <= 10)
FALSE
TRUE
approach level = 1
branch distance = count - 10 + K
if (checksum % 10 == checkdigit)
FALSE
TRUE
approach level = 0
branch distance =|(checksum % 10) - checkdigit| + K
TARGET

(b) Fitness computation for coverage of the true branch from line 19

Figure 7. A typical function for which Search-Based Software Testing may be used to generate structural test data generation, and the accompanying fitness function computation used for the coverage of a particular branch. The function is a checksum routine for registry numbers assigned to chemicals by the Chemical Abstracts Service (CAS)

is a constant added when the undesired, alternate branch is taken. The closer `count` is to being greater than 4, the 'closer' the desired true branch is to being taken. A different branch distance formula is applied depending on the type of relational predicate. In the case of `y >= x`, and the >= relational operator, the formula is $x - y + K$. A full list of branch distance formulae for different relational predicate types is provided by Tracey *et al.* [29].

The complete fitness value is computed by normalizing the branch distance and adding it to the approach level. Different functions can be used to normalize the branch distance, and these are evaluated and discussed by Arcuri [30].

IV. Future Directions and Open Problems

A. Handling the Execution Environment

One open problem with Search-Based Software Testing techniques, and Search-Based Test Data Generation techniques in particular, is lack of handling of the execution environment that the software under test lives within. Current state of the art in test data generation, for example, ignores or fails to handle interactions with the underlying operating system, the file system, network access and databases on which they may be dependent. A recent study by Lakhotia *et al.* [31], [32] cited some of these factors as sources of poor coverage with the AUSTIN Search-Based Test Data Generator. To date, search-based tools have largely generated test data for primitive types only, such as `int`, `double` and strings of characters. There has also been work dealing with dynamic data structures [33], and the `eToc` tool of Tonella [34] will generate object parameters for Java programs.

The execution environment presents non-standard challenges for Search-Based Test Data Generation approaches. Difficulties with the file system include testing code or generating code coverage for programs that check the existence of files or directories, reading and validating files, handle read/write errors or other issues such as a full file system. Programs using databases tend to include code that perform tasks such as opening a connection to the database; inserting, updating and deleting data; testing for the presence of certain values and combinations in the database, and handling concurrent updates. The underlying operating system environment may cause problems when the program is checking for the amount of available memory, using the values of environment variables, or rendering graphics to the display. Code involving network access may need to read or write values from and to a socket, check for the presence of services and so on.

Some of these issues might be dealt with by generating test data that is then copied to a file or a database, to be read back in by the program under test. In unit testing, the common solution is to use mock objects. For example, the Java method of Figure 8 involves reading information from a database. The skeleton mock objects

```
public String readPeople(MyDatabase db) {

  MyRecordset r = db.executeQuery(
      "SELECT name, age FROM people");
  String result = "";
  int num = 0;

  while (r.next()) {
    String name = r.getString("name");
    int age = r.getInt("age");
    result += num+": "+name+", age "+age+"\n";
  }

  return result;
}
```

Figure 8. Snippet of code that reads values from a database

```
public MockDatabase extends MyDatabase {

  public MyRecordset executeQuery(String query) {
    // ...
  }
}

public MockRecordset extends MyRecordset {

   public boolean next() {
    // ....
  }
}
```

Figure 9. Skeleton mock objects for the example of Figure 8

```
void readFile(MyFile f) {

  if (f.readInt() == 0) {
    if (f.readChar() == ",") {
      if (f.readInt() == 1) {
        // target
      }
    }
  }
}
```

Figure 10. Snippet of code from reading values from a file

```
public class MockFile extends MyFile {

  int readIntCall = 0;

  int readInt() {
    if (readIntCall == 0) return 0;
    else return 1;
    readIntCall ++;
  }

  String readChar() {
    return ",";
  }
}
```

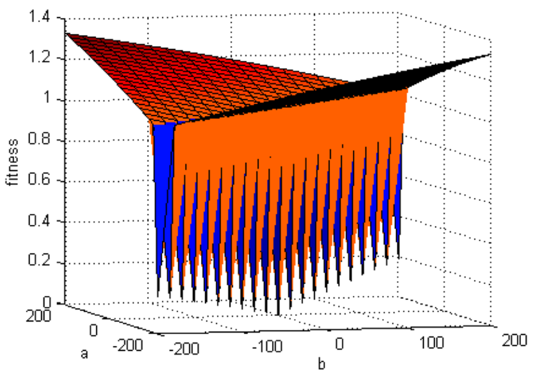
Figure 11. Mock object required for executing the target in the program of Figure 10

```
void original(double a, double b) {
  if (a == b) {
    double c = b + 1;
    if (c == 0) {
      // target
    }
  }
}
```

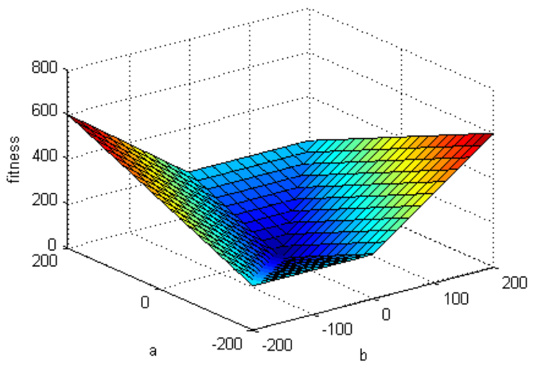
(a) Original program

```
void transformed(double a, double b) {
  double _dist = 0;
  _dist += distance(a == b);
  double c = b + 1;
  if (_dist == 0.0) {
    // target
  }
}
```

(b) Transformed version



(c) Landscape for original program



(b) Landscape for transformed version

Figure 12. Demonstrating the nesting testability transformation (taken from [35])

that might be generated by such a tool can be seen in Figure 9, which comprise the `MockDatabase` and the `MockRecordset` classes. `MockDatabase` provides a stub method for `executeQuery()` in `Database`, while `MockRecordset` stubs `next()` in `Recordset`. However, as can be seen, there is usually still a burden on the tester, in that they must provide the values that must be returned by each method in the right order, in order to test some aspect of the code.

Figure 10 is another example of where a mock object might be required. In the code snippet, an object representing a file is passed to the method. Specific values need to be read from the file in order for the target to be executed. Could these values be automatically generated using Search-Based Techniques? Or, could Genetic Programming be used to complete the mock object skeletons generated by other tools? The mock object generated for the file example of Figure 10 might look something like the code in Figure 11. The traditional structural testing fitness function might be re-used in this context. The approach level metric informs the search that a certain sequence of values is required, while the branch distance metric may help guide the Genetic Programming search to the generation of the required values.

B. Advanced Approaches to Improving Testability

Because fitness functions are heuristics, there are cases in which they fail to give adequate guidance to the search. A classic case is the so-called 'flag' problem in structural test data generation [36], [37], [38], [39], [40]. This situation occurs when a branch predicate consists of a boolean value (the 'flag'), yielding only two branch distance values; one for when the flag is true, and one for when it is false. The fitness landscape essentially consists of two plateaux, with no gradient for directing the search process, which becomes unguided and random. Harman *et al.* introduced the concept of a 'Testability Transformation' [40], [37], [39] to deal with this problem. A Testability Transformation produces a temporary version of the program under test to remove the awkward landscape feature for Search-Based Test Data Generation. Once test data has been generated, using the transformed version of the program, it can be discarded. For programs with flags, the boolean variable is removed from the conditional and replaced with a condition that leads to the flag becoming true, giving a wider range of branch distance values and making the program more amenable to search-based techniques.

Testability Transformations have also been applied to nested targets in Search-Based Test Data Generation. In the search-based approach, the branch distance for each conditional, at each approach level, is minimised one after the other. This can cause the search to be inefficient, or cause it to over-fit to early information, as is the case with the example in Figure 12a. Initially, to generate test data to reach the target, the condition '`a == b`' must be made true. It is only when this occurs that the search must then satisfy `c == 0`, which requires the input value of `b` to be -1. However, it is unlikely that this value will be generated for `b` by chance, and in making moves or mutations on `b`, the search is likely to break the previous condition, *i.e.* `a == b`. The points in the search landscape where `a == b` where `a` and `b` are not equal to -1 represent local optima in the search landscape (Figure 12c). The Testability Transformation proposed by McMinn *et al.* [41], [35] flattens the nesting structure of the program, so that all branch distances may be collected. The transformed version of the program of Figure 12a can be seen in Figure 12b. This produces a dramatic change in the fitness landscape, as can be seen in Figure 12d; local optima are removed and replaced with smooth gradients to the required test data. Note however, that the transformed version is no longer equivalent to the original program. This does not necessarily matter, so long as the test data that executes the target for the transformed version also executes the target in the original. McMinn *et al.* performed a large empirical study, the results of which can be seen in Figure 13. Due to the stochastic nature of the search algorithms, each search with each branch was re-performed a number of times to even out random variation. From this, the *success rate* is found; the percentage of runs in which test data was successfully found for the branch. While the success rate improved for the majority of branches, cases existed where the transformation caused the search to become worse.

The nesting Testability Transformation is *speculative*, in that it is likely to improve the reliability of test data generation, but may also make the search worse in certain cases. Another such speculative transformation was proposed by Korel *et al.* [42], for programs with complex data dependencies. In such situations, it may make sense to perform the search with both transformed and untransformed versions of the program. Furthermore, there is no limit to the number of potential speculative testability transformations that could be performed, each of which may represent a particular 'tactic' for improving the reliability of the test data generation process.

In addition, each transformation could be attempted in parallel as 'Co-Testability Transformations' [43], each occupying a portion of search resources. Each transformation could potentially compete for search resources, obtaining a large share of the individuals of the population of a Genetic Algorithm, for example, if that transformation led to large increases in fitness. Resources allocated to transformation that perform poorly could be removed, until the portion of the population designated to it dies out.

This is similar to the 'Species per path' approach of McMinn *et al.* [44]. In this work, the population of a Genetic Algorithm for covering a search target was split up. Each sub-population was given a different fitness function for finding test data for a branch, based on the execution of a particular path through the program. The progress for one of these searches can be seen in Figure 14. Some of the species (*e.g.* species 5) corresponded to infeasible paths, and so further improvements in fitness were not possible after a certain point. On the basis of this information, resources (numbers of individuals in a population) could be reallocated to species that were making continual improvements (such as species 7), and so speed up the discovery of test data.

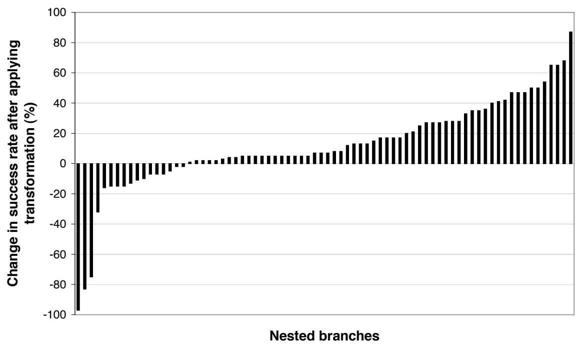


Figure 13. Results with the nesting testability transformation (taken from [35]). The transformation is 'speculative', in that while it improves the success of test data generation for the majority of branches, cases exist where the chances of test data generation are reduced

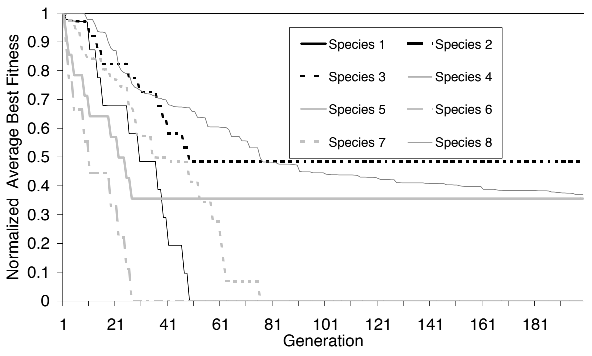


Figure 14. The progress of each species in the 'Species per Path Approach' to test data generation (taken from [44]). Some species are attempting to find test data for infeasible paths for executing the target, and so progress can only be made up to a certain point in the search process. Other species correspond to feasible paths, for which test data are successfully generated

C. Automated Oracles via Testability Transformation

Another way in which transformation may be utilized is to produce alternative versions of a program that can be used to check the original. This is similar to the idea of N-versioning [46], [47], except that each different version of the program is produced automatically, and is designed to target a particular type of fault. For example, McMinn presented a transformation designed to test the reliability of code with floating point numbers [45]. The IEEE standard used by both C and Java for representing numbers of type `double`, is incapable of representing certain numbers of finite decimal representation, such as `0.1` [48]. For example in Java, the operation `0.1 + 0.1 + 0.1` yields '`0.30000000000000004`' rather than simply `0.3` (Figure 15). Such small numerical errors may accumulate into larger errors. Additional further errors can exist, for example through premature rounding.

	Using the `double` primitive type	**Using the `java.math.BigDecimal` class**

Java statement:	`System.out.println(0.1 + 0.1 + 0.1);`	`System.out.println(new BigDecimal("0.1").add(`
		`new BigDecimal("0.1")).add(`
		`new BigDecimal("0.1")));`

Output:	`0.30000000000000004`	`0.3`

Figure 15. Comparing floating-point arithmetic in Java (version 6) using `double` compared to `BigDecimal` (taken from [45])

The transformed version of the program replaced the type of all `double` variables with variables of type `BigDecimal`, a Java class designed to handle floating-point numbers with high precision and accuracy. The output of the transformed version of the program can then be compared alongside the original. Technically, the outputs of the two programs should be identical; *i.e.*, the transformation is expected to be equivalent to the original. However if there is a discrepancy, a fault may exist in the original. It could also be that there is a problem with the transformed version of the program. In this way, the transformed version is effectively operating as a *pseudo-oracle*, as defined by Davis and Weyuker [49]; a program that has been produced to perform the same task as its original counterpart, where any discrepancy in output between the two represents either a failure on the part of the original program or its pseudo-oracle. However, the pseudo-oracles proposed by Davis and Weyuker were not produced automatically, and had to be written manually.

The pseudo-oracle transformation encapsulates some aspect of a program that should not produce any difference in behaviour when transformed, but in practice may do so in certain circumstances, and in which case may indicate a fault with the original program. Can further types of pseudo-oracle transformation be defined, and how can better fitness functions be designed to automatically reveal discrepancies between different versions of the same program? In the work of McMinn [45], differences could only be found at random, with the search was used instead to maximize the 'size' of the difference in behaviour.

D. Searching to Judge the Success of Code Migration and Refactoring

After pseudo-oracle transformation, the role of the search technique is to demonstrate a *difference* in the behaviour between the two versions of the program. Fitness functions are therefore still required that provide guidance to differences in behaviour. The idea of searching for differences between two components that are supposed to behave the same is extendable to other areas of software engineering and testing. These include searching to check that a code migration step has been performed correctly, that a particular refactoring has maintained the behaviour of the original system; and so on.

E. Minimizing Human Oracle Cost

Despite the work devoted by the software engineering community to automated oracles - in the form of modelling, specifications, contract driven development and metamorphic testing - it is still often the case that a human tester is left to evaluate the results of automatically generated test cases. However, little attention has been paid to minimizing the effort that the human tester needs to expend in this task.

Quantitative Cost. One angle on this problem is reducing the number of test cases that the human must evaluate. Harman *et al.* [50] investigated a number of fitness functions that aim to maximise structural coverage, while minimising the number of test cases need to do so. Arcuri and Fraser [51] achieve a similar effect with a 'whole coverage' approach, whereby the fitness function rewards inputs that come close to executing as many branches as possible. Higher coverage is obtained for Java programs than comparable techniques with fewer tests. Another aspect is test case size. Some automatically generated test cases may be unnecessarily long, particularly for object-oriented tests, where a sequence of test statements must be found that construct the relevant objects required for the test, and put them into the required tests. Arcuri and Fraser [52] have investigated the problem of controlling test case 'bloat' (*i.e.* unnecessary statements in tests for object-oriented programs) while Leitner *et al.* [53] have investigated minimizing unit tests using an approach based on Delta-Debugging [54], which is used to identify ineffectual statements in the test case.

Qualitative Cost. The above approaches tackle the *quantitive* aspects of the human oracle cost problem. McMinn *et al.* [55] were the first to address the *qualitative* aspects, *i.e.* how easy the scenario underpinning a test case is to understand by a tester, so that they can quickly and easily judge whether the test case succeeded or not. The example given is that of a calendar program. Usually a human would expect recognisable dates, such as 1/1/2010, as inputs. However, automatic test data generators will produce test data capable of merely executing the test goal at hand, producing very strange dates such as 4/10/-29141 and 10/8/6733; and then requiring the human tester to check that the outputted number of days between the two days is correct. McMinn *et al.* [55] propose several means of alleviating this problem:

a) Seeding domain knowledge. The starting point of any search-based approach may be explicitly set rather than been generated at random, with the intention of providing the search with some known examples or domain knowledge. This is known as 'seeding'. This initial biasing of the search tends to produce results that are in the neighbourhood of those starting points. To reduce human oracle cost, the tester could be asked to provide a few test cases of their own.

Since these test cases are likely to contain 'recognisable' data, the newly generated test data are also likely to have this characteristic. Since the programmer is likely to have run their program at least once with a 'sanity' check, the provision of human-generated test cases is not a unreasonable requirement. Indeed, the tester may wish to bias the search process with their own favourite example or corner cases.

b) Extracting information from the program. The program itself may be a rich source of information that gives clues as to the types of inputs that may be expected. For example the identifier names 'day', 'month' and 'year' imply a certain range of integers or string values, depending on the types of their variables. Further identifier analysis might be performed using the identifier extrapolation work of Lawrie *et al.* [56].

c) Re-using test data. Finally, test data may be re-used from one program to another. If a test suite exists for functions, routines or programs similar to the program under test, they could be used as the starting point for further test data generation.

One possible objection to this work is that the reduction in fault-finding capability of the test suites produced. No studies have been performed to date that show whether this is or is not the case. However, low human oracle cost test suites may be augmented with generated cases in the traditional fashion, or some trade-off sought to balance fault-finding capability with oracle cost. This is essentially a two-objective approach. Search-based approaches are well placed to handle such problems, as discussed in the next section.

F. Multiple Test Objectives

One overlooked aspect of Search-Based approaches is the ability to optimize more than one fitness function at once. This allows for the search to seek solutions that satisfy more criteria than just for example, structural coverage. The result of multi-objective search [57] is a set of Pareto-optimal solutions, where each member of the set is not better than any of the others for all of the objectives. Multi-objective search provides an advantage over traditional testing techniques that are only capable of doing 'one thing', e.g. generating test sets that cover as much of the software as possible. To date multi-objective search has been applied in Search-Based Software Testing to produce test sets that cover as much of the code as possible while also maximising memory consumption [58]. Other applications have included prioritising tests that cover as much of the software as possible whilst minimising the amount of time of the tests take to run [16]. There are several other potential application areas, including producing test sets that produce as include as much coverage (or fault finding power) as possible whilst minimising oracle cost, maximising coverage and test case diversity (in the hope of trapping more faults), and so on.

V. Conclusions

Since the cost of manual testing in practice is very high, research into automated software testing is of high concern. Search-Based Software Testing is a very generic approach in which solutions may be sought for software testing problems automatically, using optimisation algorithms.

This paper has reviewed some common search algorithms, and some of the classic testing problems to which the search-based approach has been applied. The paper has also discussed several avenues worthy of future investigation.

Acknowledgements

This paper is a brief account of author's keynote given at the 4th International Workshop on Search-Based Software Testing, in Berlin, Germany, March 2011. Although it is a single-author paper, some of ideas presented have been the result of discussions and conversations with several colleagues, including John Clark, Robert Hierons, Mark Harman, Gregory Kapfhammer, Kiran Lakhotia, Marc Roper and Mark Stevenson. I am also grateful to Yuanyuan Zhang for collating and supplying the data for Figure 1 from the SEBASE publication repository (`http://www.sebase.org/sbse/publications`).

The ideas presented in Section IV-E are the basis of ongoing work supported by the EPSRC-funded project RECOST - REducing the Cost of Oracles in Software Testing (EP/I010386/1) (`http://recost.group.shef.ac.uk`).

The author is further supported by the EPSRC grants 'Automated Discovery of Emergent Misbehaviour' (EP/G009600/1) and 'REGI: Reverse Engineering State Machine Hierarchies by Grammar Inference' (EP/F065825/1).

References

[1] W. Miller and D. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, 1976.

[2] http://www.time.com.

[3] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[4] ——, "Dynamic method for software test data generation," *Software Testing, Verification and Reliability*, vol. 2, no. 4, pp. 203–213, 1992.

[5] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels)," in *5th International Conference on Software Engineering and its Applications*, Toulouse, France, 1992, pp. 625–636.

[6] O. Buehler and J. Wegener, “Evolutionary functional testing of an automated parking system,” in *International Conference on Computer, Communication and Control Technologies and The 9th International Conference on Information Systems Analysis and Synthesis*, Orlando, Florida, USA, 2003.

[7] ——, “Evolutionary functional testing,” *Computers & Operations Research*, vol. 35, pp. 3144–3160, 2008.

[8] P. Puschner and R. Nossal, “Testing the results of static worst-case execution-time analysis,” in *Proceedings of the 19th IEEE Real-Time Systems Symposium*. Madrid, Spain: IEEE Computer Society Press, 1998, pp. 134–143.

[9] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres, “Testing real-time systems using genetic algorithms,” *Software Quality Journal*, vol. 6, no. 2, pp. 127–135, 1997.

[10] J. Wegener and M. Grochtmann, “Verifying timing constraints of real-time systems by means of evolutionary testing,” *Real-Time Systems*, vol. 15, no. 3, pp. 275–298, 1998.

[11] L. C. Briand, J. Feng, and Y. Labiche, “Using genetic algorithms and coupling measures to devise optimal integration test orders,” in *14th IEEE Software Engineering and Knowledge Engineering (SEKE)*, Ischia, Italy, 2002, pp. 43–50.

[12] Z. Li, M. Harman, and R. M. Hierons, “Search algorithms for regression test case prioritization,” vol. 33, no. 4, pp. 225–237, 2007.

[13] L. C. Briand, Y. Labiche, and M. Shousha, “Stress testing real-time systems with genetic algorithms,” in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*. Washington DC, USA: ACM Press, 2005, pp. 1021–1028.

[14] Y. Jia and M. Harman, “Constructing subtle faults using higher order mutation testing,” in *8th International Working Conference on Source Code Analysis and Manipulation (SCAM 2008*. Beijing, China: IEEE Computer Society, 2008, to appear.

[15] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, “Time aware test suite prioritization,” in *International Symposium on Software Testing and Analysis (ISSTA 06)*. Portland, Maine, USA: ACM Press, 2006, pp. 1–12.

[16] S. Yoo and M. Harman, “Pareto efficient multi-objective test case selection,” in *International Symposium on Software Testing and Analysis (ISSTA’07)*. ACM Press, July 2007, pp. 140–150.

[17] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, “Constructing test suites for interaction testing,” in *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*. IEEE Computer Society, 2003, pp. 38–48.

[18] K. Derderian, R. Hierons, M. Harman, and Q. Guo, “Automated unique input output sequence generation for conformance testing of FSMs,” *The Computer Journal*, vol. 39, pp. 331–344, 2006.

[19] N. Tracey, J. Clark, K. Mander, and J. McDermid, “Automated test data generation for exception conditions,” *Software - Practice and Experience*, vol. 30, no. 1, pp. 61–79, 2000.

[20] M. Harman and B. Jones, “Search-based software engineering,” *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[21] P. McMinn, “Search-based software test data generation: A survey,” *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[22] S. Kirkpatrick, C. D. Gellat, and M. P. Vecchi, “Optimization by simulated annealing,” *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[23] D. Whitley, “A genetic algorithm tutorial,” *Statistics and Computing*, vol. 4, pp. 65–85, 1994.

[24] ——, “An overview of evolutionary algorithms: Practical issues and common pitfalls,” *Information and Software Technology*, vol. 43, no. 14, pp. 817–831, 2001.

[25] M. Harman, “The current state and future of search based software engineering,” in *Future of Software Engineering 2007 (FOSE 2007)*. IEEE Computer Society, 2007, pp. 342–357.

[26] M. Harman and J. Clark, “Metrics are fitness functions too,” in *International Software Metrics Symposium (METRICS 2004)*. IEEE Computer Society, 2004, pp. 58–69.

[27] J. Wegener and F. Mueller, “A comparison of static analysis and evolutionary testing for the verification of timing constraints,” *Real-Time Systems*, vol. 21, no. 3, pp. 241–268, 2001.

[28] J. Wegener, A. Baresel, and H. Sthamer, “Evolutionary test environment for automatic structural testing,” *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[29] N. Tracey, J. Clark, K. Mander, and J. McDermid, “An automated framework for structural test-data generation,” in *Proceedings of the International Conference on Automated Software Engineering*. Hawaii, USA: IEEE Computer Society Press, 1998, pp. 285–288.

[30] A. Arcuri, “It does matter how you normalise the branch distance in search based software testing,” in *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, to appear, 2010.

[31] K. Lakhotia, P. McMinn, and M. Harman, “Automated test data generation for coverage: Haven’t we solved this problem yet?” in *Proceedings of the Testing: Academic & Industrial Conference - Practice And Research Techniques (TAIC PART 2009)*. IEEE Computer Society, 2009, pp. 95–104.

[32] ——, “An empirical investigation into branch coverage for C programs using CUTE and AUSTIN,” *Journal of Systems and Software*, vol. 83, pp. 2379–2391, 2010.

[33] K. Lakhotia, M. Harman, and P. McMinn, “Handling dynamic data structures in search based testing,” in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2008)*. Atlanta, USA: ACM Press, 2008, pp. 1759–1766.

[34] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the International Symposium on Software Testing and Analysis*. Boston, USA: ACM Press, 2004, pp. 119–128.

[35] P. McMinn, D. Binkley, and M. Harman, "Empirical evaluation of a nesting testability transformation for evolutionary testing," *ACM Transactions on Software Engineering Methodology*, vol. 3, 2009.

[36] L. Bottaci, "Instrumenting programs with flag variables for test data search by genetic algorithm," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*. New York, USA: Morgan Kaufmann, 2002, pp. 1337 – 1342.

[37] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer, "Improving evolutionary testing by flag removal," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*. New York, USA: Morgan Kaufmann, 2002, pp. 1359–1366.

[38] A. Baresel and H. Sthamer, "Evolutionary testing of flag conditions," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*. Chicago, USA: Springer-Verlag, 2003, pp. 2442 – 2454.

[39] A. Baresel, D. Binkley, M. Harman, and B. Korel, "Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*. Boston, Massachusetts, USA: ACM, 2004, pp. 43–52.

[40] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.

[41] P. McMinn, D. Binkley, and M. Harman, "Testability transformation for efficient automated test data search in the presence of nesting," in *Proceedings of the UK Software Testing Workshop (UKTest 2005)*. University of Sheffield Computer Science Technical Report CS-05-07, 2005, pp. 165–182.

[42] B. Korel, M. Harman, S. Chung, P. Apirukvorapinit, and G. R., "Data dependence based testability transformation in automated test generation," in *16th International Symposium on Software Reliability Engineering (ISSRE 05)*, Chicago, Illinios, USA, 2005, pp. 245–254.

[43] P. McMinn, "Co-testability transformation," in *Proceedings of the Testing: Academic & Industrial Conference: Practice And Research Techniques (TAIC PART 2008), Fast Abstract*, 2008.

[44] P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to search-based test data generation," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006)*. Portland, Maine, USA: ACM, 2006, pp. 13–24.

[45] P. McMinn, "Search-based failure discovery using testability transformations to generate pseudo-oracles," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2009)*. Montreal, Canada: ACM Press, 2009, pp. 1689–1696.

[46] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault-tolerance during execution," in *Proceedings of the First International Computer Software and Application Conference (COMPSAC '77)*, 1977, pp. 149–155.

[47] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. 11, pp. 1491–1501, 1985.

[48] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 21, no. 1, pp. 5–48, 1991.

[49] M. Davies and E. Weyuker, "Pseudo-oracles for non-testable programs," in *Proceedings of the ACM '81 Conference*, 1981, pp. 254–257.

[50] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Proceedings of the 3rd International Workshop on Search-Based Testing*. IEEE digital library, 2010.

[51] G. Fraser and A. Arcuri, "Whole suite test data generation," in *International Conference On Quality Software (QSIC 2011)*, to appear, 2011.

[52] ——, "It is not the length that matters, it is how you control it," in *IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, 2011.

[53] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Automated Software Engineering (ASE 2007)*. Atlanta, Georgia, USA: ACM Press, 2007, pp. 417–420.

[54] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, pp. 183–200, 2002.

[55] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proceedings of the 1st International Workshop on Software Test Output Validation (STOV 2010)*. Trento, Italy: ACM, 2010, pp. 1–4.

[56] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *International Working Conference on Reverse Engineering (WCRE 2010)*. IEEE Computer Society, 2010, pp. 3–12.

[57] K. Deb, "Multi-objective evolutionary optimization: Past, present and future," in *Proceedings of the Fourth International Conference on Adaptive Computing in Design and Manufacture (ACDM'2000)*. University of Plymouth, UK: Springer, London, 2000, pp. 225–236.

[58] M. Harman, K. Lakhotia, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*. London, UK: ACM Press, 2007, pp. 1098–1105.