AN ABSTRACT OF THE DISSERTATION OF

Ali Aburas for the degree of Doctor of Philosophy in Computer Science presented on November 07, 2016.

Title: Enhancing Search-Based Techniques with Information Control Dependencies

Abstract approved: _____

Alex D. Groce

Software testing is a very important task during software development and it can be used to improve the quality and reliability of the software system. One potential way to reduce the cost and increase the efficiency of software testing is to generate test data automatically. Search-based approaches successfully generate unit tests for object-oriented programs, like Java. However, challenges, such as the large size of the search space, and the presence of complex predicates target branches, negatively affect the approaches, and, thus, cannot achieve high structural coverage for certain programs.

The aim of this thesis is to propose enhancement techniques to improve the effectiveness of search based testing approaches and address the challenges posed by the object-oriented programs. Rather than randomly generating a sequence of method calls, our ongoing work is to focus on using static analysis to define the hidden data dependencies on predicates target branches (i.e., uncovered branches) and exploit method dependence relations (MDR) approach to precisely identify the method/constructors and its parameters. This method dependence information is employed to reduce the search space and

used to guide the search toward regions that lead to full (or at least high) structural coverage.

Enhancing Search-Based Techniques with Information Control Dependencies

by

Ali Aburas

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented November 07, 2016
Commencement June 2017

Doctor of Philosophy dissertation of Ali Aburas presented on November 07, 2016.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Ali Aburas, Author

ACKNOWLEDGEMENTS

First and above all, I praise God (ALLAH), the almighty for providing me with good health and wellbeing to complete this thesis. I would also like to offer my sincere gratitude to my advisor Dr. Alex Groce for his endless support and advice throughout my Ph.D study, and for his patience and motivation.

Many thanks to my sisters and brothers who always stand by my side. I also warmly thank and appreciate my elder brother Hussein who brought me up like a father. He is a great inspiration to me. I want also to express my gratitude and deepest appreciation to my wife, Nedda, who always prayed for my success. I understand it was difficult for you, therefore, I can just say thanks for everything and may ALLAH give you all the best in return. Finally, I appreciate the financial support of the Libyan Ministry of Higher Education and Scientific Research during my Ph.D study.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

1.1 Motivation

Software testing is a very important task during software development and it can be used to improve the quality and reliability of the software system. Unfortunately, software systems are becoming larger and more complex. Therefore, manually generating test data of these systems is expensive, time consuming, and error prone[68]. One potential way to reduce the cost and increase the efficiency of software testing is to generate test data automatically.

Object-oriented software systems are heavily depended on the internal state of both receiver and argument object instances [14]. Objects are instances of a class, and they significantly increase the complexity of testing [59]. To alleviate the burden in object-oriented unit testing, a number of automated test generation tools generate test inputs for a unit (e.g. a class, or a method) [22, 31, 59, 79]. These test inputs are in the form of method call sequences, where (1) desirable primitive method arguments values(e.g., integer) are automatically generated, and (2) desirable method argument objects and receiver objects (e.g., a container class such as stack and list) are automatically generated via a sequence of method calls [59].

When automatically performing unit test generation for object-oriented programs like Java, the primary goal is to achieve a high or full code coverage (such as branch coverage), of the class under test (CUT), which gains confidence in the CUTs quality and functionality [94]. However, automatically generating test inputs for a unit test requires

a desirable sequence of method calls that create and put objects into particular states. Yet, how to generate these desired objects to achieve high code coverage is a challenging task which hinders the automated test generation approaches [14]. A study shows that failing to generate desirable objects is the main cause of low code coverage for automated test generation tools [59].

There are many automated test input generation approaches, including random testing, symbolic execution, and search based approaches. Random testing approaches [22, 79] are easy to implement, applicable, and the fastest in execution [105]. 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 [20]) explore path conditions in the program under test and collect constraints from 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. Finally, the search based approaches employ meta-heuristic optimization techniques, such as Genetic Algorithms, and use a fitness function that guides the search toward better solutions. Thus, the effectiveness of the search algorithms is improved as long as the fitness function distinguishes between better and worse solutions [13].

Search based test generation approaches (e.g., EvoSuite [31]) have already been shown to be effective for generating test data that achieves high code coverage and reveals failures for object-oriented programs [1, 13, 16, 31, 50]. 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 method calls (to optimize the solutions) can be a challenge due to the large size of the search space [48]. In addition, the efficiency of the search based approaches decreases for programs that have branch predicates using boolean or string constants, i.e., the flag problem [68]. In this case, the resulting fitness function only produces two different values. Therefore, no heuristic can be defined to guide the search, since the fitness function landscape contains plateaus [13] [68]. Finally, the usage of non-primitive data types as parameters of the methods, such as interfaces and object references, reduce the effectiveness of the search based test data generation because it needs to decide which class candidate to instantiate [68].

In response to this context, this thesis proposes scalable automatic techniques to automate the test input generation process. The techniques developed in this thesis utilizes the static techniques to exploit program dependencies (e.g., control and data dependencies) to generate desirable test inputs and strengthen structural coverage testing.

The statically-identified dependence information enhances two search-based approaches: a genetic algorithm (GA) and memetic algorithm (MA). In particular, we introduce two novel search-based approaches, called GAMDR [2] and MAMDR [1], to handle large search space. We use search-based approaches [15] [16] and augment with method dependency relations (MDR) [104] to guide the search to generate relevant sequences of method calls that achieve high code coverage for unit-class testing.

1.2 Research Objectives

The overall objectives of this research are to investigate and establish approaches that can effectively overcome some of the limitations of the current state-of-the-art search

based testing techniques for object-oriented programs. Particularly, we want to propose a novel automated search based testing technique which is able to generate more high-covering test data.

Although, we implement our enhancement techniques in Java as an example of object-oriented language to prove the concepts, our approach, however, is applicable in general to any typed object-oriented language.

1.3 Scope of Research

In our view, state-of-the-art search based approaches mostly focus on randomly-created sequences of method calls and, thus, neglect to support of object-oriented language features such as encapsulation, inheritance, and polymorphism. They fail to generate test data, and therefore, fail to produce desirable object states and the sequences of method calls for non-primitive method arguments or receiver [59].

In response to this context, this thesis mainly covers the following topics in the software testing area, search-based software testing of object-oriented systems, and structural coverage testing.

1.3.1 Search-Based Software Testing of Object-Oriented Software

In this thesis, we focus on incorporating knowledge about a program into the search process for alleviating the current limited level of dealing with object-oriented language features. Thus, one focus of our research is to propose techniques that deal effectively with object-oriented programming concepts. At a high level, we see that we can use variety of analysis techniques to detect data dependency relations within the class un-

der test, and then adapt search based approaches to take advantage of this information. If such dependency information is available, our techniques will help the search to strengthen structural coverage and find better solutions than state-of-the-art search based approaches.

In essence, the goal of this research is to precisely identify a root cause that prevents achieving high structural coverage and steer search based test data generation in a directed, automated manner toward effectively and efficiently exploring the search space. Therefore, this leads to another focus area, which is to investigate different techniques that extend the search based techniques to generate input data that simultaneously deals with both complexity and scalability problems. Hence, the suggested techniques should be simple, scalable, and effective in practice when, combined with a search based technique.

We believe that precise knowledge of data dependence can be used to guide the search to only explore regions in the search space that help to generate required input data [100]. This tends to increase the efficiency of the search in general, and cover as much code as possible even in the presence of complex predicates in particular.

With these objectives in mind, it is important to demonstrate the effectiveness of our techniques and its efficiency at improving structural coverage. Hence, we will empirically validate the proposed techniques by designing a set of experiments to compare their capability to improve structural coverage with the state-of-the-art search based approaches.

1.3.2 Structural Coverage Testing

The main activity in software testing is to generate test cases that aim to achieve a high degree of some adequacy criterion. An adequacy criterion is very helpful for testers and plays two fundamental roles in software testing [106]. First, it can be used as a stopping rule that indicates how sufficiently the testing has been performed. Second, it can be used as a measurement of degree of test-suite quality. Zhu *et al.* [106] define three different test adequacy criteria (1) structural coverage, (2) fault-based, and (3) error-based adequacy criteria.

The structural coverage criterion (branch, path, and statement coverage) emphasizes the need to exercise particular component in the source code of the class under test [106]. For example, the branch adequacy criterion requires that all the branches in the class under test are exercised during testing [106]. This information can be used to indicate the degree to which of the source code of the class under test is covered during the software testing. Although a high code coverage degree does not make test cases more effective in finding faults, but it offers a high confidence about the class quality and reliability [52].

This thesis focuses on exploring effective search-based techniques to automatically generate test cases that cover many branches as possible, and ideally we are going to compare the results in terms of the achieved branch coverage. The branch coverage was chosen because it has been shown to be the most commonly considered criteria in the existing literature [52].

1.4 Hypotheses and Research Questions

The underlying premise of this research study is that the search must first identify the regions in the search space that contain feasible test data, and then explore them to generate required sequences of method calls such that as much code as possible is covered [59].

The overall objectives of this research formulate a hypothesis that expresses the intent of what we are setting out to achieve. At a high level, our research hinges on the following hypothesis:
incorporating the search based test data generation process with data dependency of the predicates branches can produce test data with high branch-coverage for object-oriented programs.

To investigate this hypothesis, three research questions are considered:

RQ1: To what extent does the presence of dependencies in the complex predicates hinder the search for test object-oriented programs using evolutionary testing?

RQ2: How can data dependency analyses be managed in the SBST techniques such that the structural coverage criteria and scalability are maximized?

RQ3: How effective are the test cases generated using our techniques?

1.5 Contributions of The Thesis

The major contributions of this thesis to existing knowledge can be as follows:

1. GAMDR, a fully automated featured search-based software testing approach for

Java.

2. A description of how a light-wight static approaches from several researches on test data generation are incorporated into the design of GAMDR.

3. Different enhancement techniques, such as seeding and local search, are proposed to improve the effectiveness and efficiency of the search process.

4. An empirical study which evaluates the level of code coverage that can be obtained using all the proposal presented in this thesis. To this end, we select several open source Java projects, and compare the performance of our approaches against the state of the art in test generation, considering random testing and search based approaches.

Chapter 2: Background and Terminology

This chapter contains common concepts used throughout the thesis, and they have been added to make the thesis self-contained. The chapter starts by describing the various features of object-oriented programming and how these features affect testing of object-oriented systems. This chapter also defines the difference between procedural and object oriented testing. Finally, the chapter provides background information on the most of the topics covered in our technical approaches.

2.1 Object-Oriented Programming

What then, is object-oriented programming?. Booch [19, p. 35] defines it as the following:

Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

There are three important parts to this definition: object-oriented programming (1) uses objects; (2) each object is an instance of a class; and (3) classes are related to one another via inheritance relationships. In other words, a programming language is considered object-oriented if it has mechanisms that support classes and data abstraction, encapsulation, inheritance, and polymorphism [19, p. 35]. The following sections describe

these concepts.

2.1.1 Data Abstraction (Objects and Classes)

Abstraction is an exceptionally powerful technique that we as humans use for dealing with complexity [19, p. 38]. In other words, an abstraction focuses on the outside view of an object, and it takes out unnecessary details and separates an objects necessary behavior from its internal implementation [19, p. 46].

An object-oriented program is object-based, and composites of interacting objects that act upon themselves via invoking a sequence of method calls [19, p. 38]. The state of an object depends on the data that are stored within its fields, and the behavior of the object is influenced by its state.

In an object-oriented program, like Java, a class is used as the basis of how to instantiate objects [103, p. 15]. Thus, a class is an abstract data type and it typically includes attributes, and methods. The attributes are variables that represent the state of an object, and the methods are operations that typically perform different operations upon an object, such as altering, accessing the state of an object, or creating a new object. Thus, the state of an object is affected by the order of method calls upon the object.

2.1.2 Encapsulation

Encapsulation is one of the primary advantages of using objects, and it is often achieved through information hiding [19, p. 46]. As Weisfeld suggests [103, p. 19], in good object-oriented design, an object should only reveal the attributes and behaviors that other

objects must have to interact with it. For example [103, p. 19], an object that calculates the square of a number must provide a method to obtain the result, and the details of its internal implementation remain hidden from all other objects.

In practice, an object-oriented program language (e.g., Java) offers control over the visibility of member objects . Specifically, the methods and attributes of a class can be declared as *public*, *private* and *protected* to mark its visibility [19, p. 51]. A class member (i.e. a data type or method) declared *public* is visible to all other objects of the application, and they can directly access it, regardless of which class declares it. A class member declared *private* is fully encapsulated, and only the specific objects of the class that declares it can access it. Finally, when a class member is declared as *protected*, only the objects of the class that declares it and objects of all sub-classes of declaring class can access it.

2.1.3 Inheritance

Object-oriented programming allows a class to inherit attributes and methods from other classes [103, p. 22]. The most important reason underlying the usage of inheritance is that defining hierarchy relationships between classes not only facilitates code reuse, but also organizes classes to inherit commonly used state and behavior from other classes.

Booch [19, p. 56] argues that inheritance is the most important "is a"hierarchy relation, and it is an important element in object-oriented systems. For example, a cat and dog are part of the mammal class. All mammals have some common attributes, such as eye, and hair color, as well as some behaviors (i.e., methods), such as growing hair, and generating heat. Without the use of hierarchies, the code for cat and dog would need to define its characteristics explicitly. A better way to avoid this redundancy is to use

inheritance, the cat and the dog classes need only to include any attributes or methods that make them unique within their classes, and all common attributes and methods could be moved up and declared in the class mammal [103, p. 22]. This is known as a *subclass* of cat and dog, where mammal is referred to as cat's and dog's *superclass*.

Inheritance thus expresses a hierarchy of abstractions, in which a subclass redefines the existing structure and behavior of its superclasses [19, p. 57]. As a result, when the subclass object is instantiated, it contains everything in its own class, as well as everything from the superclass. For instance, the dog and cat are more precisely specified mammals, they inherit all of the attributes and methods from the mammal class. As a result, when the dog or the cat class object is instantiated, it contains everything in its own class, and it inherits its general attributes from the mammal class [103, p. 23].

2.1.4 Polymorphism

Polymorphism means different forms, and it is tightly coupled to inheritance [103, p. 25]. It allows a method to evoke different behavior in different objects. For example, consider a superclass called Shape has the behavior called Area, and three subclasses Circle, Square, and Rectangle all inherit directly from Shape. Even though, Shape has a Area method, subclasses, such as Circle, and Square, override this method and provide their own actual implementation of Area. The actual method to invoke is identified at run time as it depends on the type of shape. Invoking the Area method on a Circle object calculates area of a circle, and invoking the Area on a Square object calculates area of a square [103, p. 172].

Polymorphism is a feature that allows one generic interface to be used to respond to some common set of behaviors (i.e., methods) in different ways. This is the essence

of polymorphism, a single object like Shape can have more than one form, and when a message is sent to different objects, the different objects exhibit different behavior [103, p. 155].

2.1.5 Java Generics

In Java, generics enable types (e.g., Integer, Double, or String) to be parameters when defining classes and methods, such that the same code can be called with arguments of different types [80]. It is very similar to formal parameters used in class and method declarations. However, the inputs to formal parameters are values, while the inputs to type parameters are types [32].

For example, generic concepts can be applied to create a generic method for sorting an array of objects, then the generic method can be invoked with Integer arrays, Double arrays, or String arrays, to sort the array elements. In a nutshell, generics are a feature that improves code reusability in which the same code can be reused with different inputs.

2.2 Object-oriented Features that Affect Testing

The object-oriented paradigm offers numerous benefits in software development by increasing software reusability, reliability and extendibility. Although features such as encapsulation, inheritance, polymorphism and generics eliminate some problems typical of procedural programs [103, p. 6], they introduce new challenges in software testing [61].

In the following we consider these features separately to identify the different problems each of them can introduce in software testing.

- *Encapsulation* is used to hide some of an object's attributes and methods from other objects. The object's attributes, thus, can only be accessed through a sequence of specific method calls. Because of the hiding information, the observation of the state of an object can become very hard [39]. The hidden state, which is referred to as the *State Problem* [72], poses a serious challenge to the object-oriented software testing.

- *Inheritance* allows a class (i.e., subclass) to share attributes and methods from preexisting classes (i.e., superclasses). Even though this feature provides the idea of reusability and extendability, it opens the issue of retesting [39]. Methods inherited from superclasses should be retested in the context of subclasses [61].

- *Polymorphism* is the capability of similar objects responding in different ways to the same message. In other words, it allows a method exhibits different behavior in similar objects. Although polymorphism helps make object oriented programs more flexible and reusable, it introduces undecidability in program testing because the exact implementation of a method cannot be known until runtime [39].

- *Generics* allow programmers to parameterize classes and methods with types, such that the method can be called for different types [32]. While generics prevent code duplication, this feature makes it difficult to know what is the exact type for generic type parameters [32]. This is a particularly serious problem for automated test data generation because it will blindly attempt to instantiate a new object to satisfy any generic parameters [32]. In fact, this problem is quite common in Java, as the Java compiler removes anything that was declared generic on the source code, such that any generic parameter is converted to type Object in bytecode [81].

On the whole, it is not possible to test a single class in isolation [84]. A test program for object-oriented software typically consists of an interactions of methods from multiple classes, which brings objects to complex and different states [94]. Thus, the space of potential inputs of a class under test can be huge to systematically and automatically explore.

2.3 Procedural Versus Object-Oriented Programming

Despite automated test generation has received a great deal of attention, automatically generating sufficient test inputs still remains a challenge task [94]. Lots of early work has been mainly focused on procedural programs such as C programs [68], but recent research has been focused on generating inputs for object-oriented programs such as Java [14][22][79][94].

Generating test inputs for object-oriented programs is more difficult and complicated than for procedural programs. In procedural programs, for example, the data and the operations (i.e., procedures, functions, and subroutines) that manipulate the data are separated [103, p. 19]. As a result, procedural programs testing typically require generating input of values for arguments of procedure or function under test as well as global variables. On the other hand, in object-oriented programs the data and the operations (called methods), which manipulate the data, are both encapsulated in the object [103, p. 20]. Thus, object-oriented testing not only requires generating primitive inputs, but also requires generating desirable object instances for both receiver object and arguments of method under test.

Procedural programs, also, are typically full units whose private operations are not modified by outside programs [39]. As noted in previous sections, object-oriented pro-

grams allowing the definition of private attributes and methods can be accessed only through the class itself and its subclasses. This restricted accessibility means that it is more difficult to observe an object's state. Testing is therefore more difficult because the only way to influence and observe an object's state is through a class's or other class's methods. In addition, an object could have attributes of primitive types and other object types which increases the complexity and size of the search space. Because of these difficulties, the automatic generation of test data is still an open problem in object-oriented software testing.

2.4 Test Generation

Test data generation is both one of the most important tasks and critical challenges in software testing since it can greatly reduce the effort and cost needed during the testing process [6]. It comes as no surprise that in the past few decades, a significant number of different techniques for automatic test case generation have been developed and investigated.

This section considers various automation techniques for test data case generation, including: (1) random testing, (2) symbolic execution, and (3) search-based testing.

2.4.1 Test data generation by random testing

Random testing, an approach in which the process of generating test inputs is simply performed randomly, has been shown to be a remarkably effective and simple technique for automated test input generation [7]. In random testing, the input domain of the program under test is defined, test inputs (i.e., test cases) are randomly selected from

this domain, and then the program structure is executed and the output of the program is observed for the test inputs [47]. The most distinguishing features of random testing are its ability to generate large numbers of test cases and its scalability to large software programs [79]. This is because random testing randomly generates test cases and is often without prior information about the complexity and size of the program under test [12].

Several approaches have been proposed that mainly focus on improving the effectiveness and capability of the random test technique for fault detection [78]. One example of a proposed approach is adaptive random testing (ART) [21]. ART focuses on enhancing the fault-detecting effectiveness of random testing by introducing a certain level of control over how test inputs are evenly selected across a program's input domain [78]. Ciupa *et al.* [23] also suggest adaptive random testing for object-oriented programs (ARTOO). The approach is based on the program's notion of *object distance* and applies ART approach [21] for testing object-oriented programs. Another well-known and state-of-the-art random testing approach for object-oriented programs is Randoop [79], which uses feedback information to generate test inputs that explore different a program under test executions. Additionally, Groce *et al.* [44] propose a random testing approach that is based on using a "swarm"that creates test inputs by using randomly generated configurations that omit randomly selected features. This feature omission leads to better exploration of the search space of a program under test and improves fault-detection [44].

Random testing is most often criticized because it typically fails to produce particular test inputs to trigger unanticipated behaviors and fails to cover all of the testing targets [47]. To illustrate, consider a program under test that takes a single integer input x and contains a conditional statement such as $if(x == 0)$. In random test generator , the probability of randomly generate the required value of x to cover this branch is extremely

low [12].

2.4.2 Test data generation by symbolic execution

King proposed symbolic execution in 1976 [63] that contrasted with random testing by statically analyzing a program's source code to generate test data [6]. Symbolic execution has been used for a number of applications in software testing, such as the automatic generation of test data to improve branch coverage and detect software errors [20][37][62], fault localization [83], and regression testing [88]. A number of different tools for symbolic execution has been developed for symbol execution. For example, a well-known symbolic execution tool is PathFinder [99], which automatically generates inputs to achieve high structural coverage of Java programs. Klee [20] and CUTE [90] are symbolic execution tools for C language. Finally, Pex [95] targets .NET languages.

The key idea behind symbolic execution is to replace concrete inputs of the program with symbolic variables that represent all possible values [6]. During symbolic execution, the constraints on inputs in branch statements are collected. A set of path conditions is modified to record the new constraints on the symbolic values [78]. The path condition is a boolean formula and consists of algebraic expressions and conditional operators [90]. Symbolic execution uses constraint solvers (e.g., Z3 [25]) to produce concrete values that make each path condition true [78]. The goal is to generate concrete values for inputs that execute every feasible path condition [90].

Although the symbolic execution technique offers an automated mechanism to explore all feasible executions of a program's paths, the technique suffers from significant limitations when it is applied to large, real programs [6]. One of the main limitations of this technique is its inability to symbolically execute all program paths, also known

as a path explosion [6]. Another limitation is that it cannot handle constraints involving floating point variables [6]. Additionally, symbolic execution is not able to detect the number of iterations of loop programming structures because loops dramatically increase the number of paths [68]. These problems have been addressed to improve the technique's usefulness on real world software testing programs.

Researchers in software testing have proposed different techniques to partially alleviate the problems just described. For example, dynamic symbolic execution (DSE), such as DART [38], or concolic execution, such as CUTE [90], attempt to overcome the problem of solving complex constraints. The idea is to simplify a path condition that a constraint solver cannot handle by substituting selected expressions in the path condition with runtime values. Another approach is to hybridize the DSE and search-based approaches to solve floating point computations [6]. Inkumsah and Xie [56] introduce a hybrid tool, called EVACON, and report the first results for a combined DSE and a genetic algorithm. Despite a large body of work has shown the benefits of symbolic executions in automatic test data generation, more research is needed to provide more effective general solutions [6].

2.4.3 Test data generation in search based software testing (SBST)

The aim of search based software engineering (SBSE) is to use a variety of metaheuristic search techniques to automate many human-based software engineering activities [54]. Metaheuristic techniques, such as genetic algorithms, have been proposed in different areas to automate software engineering processes, e.g., test data generation, model clustering, and cost/effort prediction [24].

One major area where SBSE has been much interest is Search-based software testing

(SBST) [54]. SBST has been successfully applied in a wide range of software testing problems such as object-oriented programs [13][97][31], web applications [4], and aspect-oriented programs [50]. SBST is the process of generating test cases utilizing search-based algorithms to find test data. SBST uses fitness functions to capture the improvement of the search process [6]. The fitness function is heuristic and used by search to both measure and compare solutions and direct the search into potentially promising areas of the search space to reach the global optimum [68][52]. The main search-based algorithms that have been applied to automate the test input generation process in software testing, including hill climbing, alternating variable method, and genetic algorithms [69].

What distinguishes SBST from other test input generation techniques is the process of finding test data itself. In contrast to symbolic execution, for example, SBST is able to handle different primitive data types such as floating-point and integral numbers [54]. In addition, SBST is widely applicable because any test objective, in principle, can be transformed into a fitness function [54]. However, despite the large body of work on SBST [52], there are still several limitations that have been impeded the wide acceptance of these techniques. McMinn [69] discussed potential research areas to improve various aspects of the SBST, including: handling the size of the search space, handling the execution environment, and hybridizing SBST with DSE to improve testability.

This section describes some SBST algorithms that have been applied in software test data generation and includes: (1) local search algorithms, (2) evolutionary algorithms and (3) memetic algorithms. In addition, the fitness function is described and the concept of domain-input reduction is explained.

2.4.3.1 Local Search Algorithms

Local search algorithms operate and aim to improve a single solution at a time by moving to local neighbor solutions of better fitness [69]. Empirical studies have shown that this search technique can be very efficient in practice [53].

With local search algorithms only the neighborhoods of one solution are considered and the fitness function is used to evaluate possible moves within the search space to reach a local optimum [53]. Often, the search can easily yield a local optimal solution but not represent a globally optimal solution. To prevent trapping the search in a local optimum, various strategies are used [68]. For example, the search is restarted with a new random solution in order to find potentially better solutions and enable the search to explore a wider region of the search space [54].

Here, we provide a brief review of two local search techniques that have been used in software test data generation, namely Hill Climbing (HC) and Alternating Variable Method (AVM).

1. **Hill Climbing (HC)** is a simple local search based technique that is both effective and easy to implement [53]. HC usually starts with an initial solution chosen randomly from the search space. At each iteration, the neighbors to the current solution are evaluated. When a fitter neighbor solution is found, it replaces the current solution. The neighbors of that solution are then evaluated to look for fitter neighbors. The process continues until no fitter neighbor is found for the current solution. At this time the search has reached a local optimum [68]. HC attempts to escape local optima by continually restarting the search. HC is often restarts as many times as computing resources allow [54].

2. **Alternating Variable Method (AVM)** is a similar technique to HC developed

by Korel [64]. The problem with HC is that can get trapped at local optima. AVM solves the problem to some extent by trying to locally optimize each input variable in isolation. This helps the search to move to the next variable if a local optimum is reached [60]. The AVM has been shown to be effective and efficient for covering branches for C programs [53].

The AVM initializes all the variables with random values. Then, it starts with exploratory moves on the first variable by increasing or decreasing its value a small amount. In the case of integral types, for example, an exploratory move starts at +1 or -1. If the changes improve the fitness function, larger changes called pattern moves are made. For example, in the case of integers, the search tries +2 then +4 [34]. A series of pattern moves is made in the same direction as long as the fitness function is improved by each pattern move [64].

When there are no further improvements in the fitness function, the search goes back to the exploratory moves on the same variable. Once there are no further improvements of the variable value, the search moves to consider the next variable for an exploratory move. If the entire set of variables have been explored, the AVM restarts from another generated input values until the computing resources is exceeded(e.g., the number of fitness evaluations) [55].

2.4.3.2 Evolutionary Algorithms

In contrast to local search algorithms, evolutionary algorithms are considered to be a global search because they simultaneously operate with more than one candidate solution [54]. They are based on the idea of genetics and evolution where new and fitter sets of the

candidate solutions, often called individuals or chromosomes, are created by combining portions of fittest candidate solutions [68]. Genetic Algorithm (GA) is probably the most commonly applied technique in evolutionary algorithms [68]. A standard GA algorithm for testing can be seen in Figure 2.1.

```
1.  P ← Randomly generate initial population
2.  While criterion is not met do
3.       Evaluate fitness of each individual in P
4.        I1, I2 ← Select two parents from P
5.        O1, O2 ← Crossover parents to I1, I2 form new offspring
6.        P' ← Mutate O1, O2
7.        P ←P'
8.  End While
```

Figure 2.1: The genetic algorithm applied in testing

GA starts with a random initial population of individuals and then the algorithm enters evolutionary iterations in the following order: 1) each individual is executed and its fitness is computed; 2) individuals with a higher fitness value are selected for populating the next generation; 3) a recombination operator is applied by taking two parent individuals and producing two new offspring; 4) after recombination, a mutation is applied, which produces small random changes to the offspring; and 5) new offspring fill the population of the next generation. The evolution is performed until a termination criterion is met, e.g., a 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 fittest individuals without any modification.

The individual length, population size, crossover, and mutation probability values in GA are referred to as GA parameters. Additionally, selection, crossover, and mutation are referred to as GA operators.

There is a subset of evolutionary algorithms, called Genetic Programming (GP), that shares many characteristics with GA, e.g., the operators of selection, reproduction, and mutation. However, the difference between the two is the representation of the individuals. In GP, individuals are normally represented as tree-structures [84][100].

2.4.3.3 Hybrid Evolutionary Algorithms

Hybrid evolutionary algorithms intrinsically integrates evolutionary algorithms with other techniques, such as local search algorithms, to improve the performance of the evolutionary algorithm such as speed of convergence [45]. In the following, we provide a brief description of some of techniques that have been incorporated with the evolutionary algorithm.

1. **Hybrid Approaches Incorporating Local Search Algorithms.** Hybridization between evolutionary algorithms and local search algorithms is known as Memetic Algorithms (MAs)[77]. On a high-level view, a genetic algorithm (GA) is used to search for more globally optimal solutions and a local search algorithm (e.g., HC) is applied to improve each solution at the end of each generation [53]. As a reported in the literature, MAs have been successfully applied to testing and showed better performance than evolutionary algorithms and local search algorithms [13][16][34][53].

2. **Hybrid Approaches Incorporating Dynamic Symbolic Execution (DSE).** The integration of evolutionary algorithms and dynamic symbolic execution has in recent years contributed to a large number of hybrid evolutionary approaches [6]. Evolutionary algorithms (e.g., GA) scales well and can handle floating point

computation well, but it may struggle to generate specific values to cover difficult branches [36], while DSE uses constraint solvers to precisely calculate the exact input values, but it may fail to solve floating point constraints [6]. This led several authors in the literature to propose a combination of GA and DSE to deliver the best of both approaches and produce better results in testing than individual approach [36] [56][67].

2.4.3.4 Fitness Functions

Fitness functions are heuristics and a fundamental part of any search algorithm. They reward individuals to guide a search toward fitter individuals and promising areas of the search space in the hope of finding a global optimal solution. In the context of structural criteria such as branch coverage, several fitness function have been proposed, and the most popular fitness function used to find test data to cover a target branch integrates two metrics, the approach level and the branch distance [101]. (More details can be found [8][68]).

1. **Approach Level** is usually represented with an integer and used to show how many of the conditional statements were not executed by a particular input to reach the target branch [8].

2. **Branch Distance** is used to solve the constraints of the predicates in the control flow graph. The branch distance computes the difference between a predicate value and a data input to execute the branch that leads to the target branch (see Tracey et al. [98] for more details). For instance, for predicate $i<10$ when the value of i is 2, then the distance to cover the false branch is $x = 10 - 2$ [36]. However, the branch

distance has to be normalized to $\frac{x}{(x+1)}$ [8], where x is the branch distance. This normalization guarantees that approach level is always more important than branch distance. In practice, to evaluate the fitness function the CUT is instrumented at bytecode level [8].

2.5 Conclusion

This chapter started by providing a basic understanding of the principles and the description of various features of object-oriented programming (Section 2.1). It was followed by a brief discussion of the challenges that object-oriented features impose for testing (section 2.2). Section 2.3 presented the differences between the object-oriented and procedural programming. Finally, several approaches and automated test input generation techniques were presented, and their limitations were analyzed (Section 2.4).

Through this critical literature review, we concluded that search-based test data generation is an active research area, leading to several recent searching techniques.

Chapter 3: Related Work

In this chapter, we discuss the most closely related Search Based Software Testing (SBST) techniques, with particular attention paid to their strengths and weaknesses. In addition, the impact of search space reduction and the seeding to improve the performance of SBST approaches for testing object-oriented programs is explored.

3.1 Search Based Unit Testing

In 1976, Miller and Spooner [75] were the first to propose a simple search technique that deals with generating floating-point test data [69]. Little progress was made until 1990 when Korel [64] extended Miller and Spooners research and developed a search technique. This search technique targets each branch predicate in turn, known as the Goal Oriented Approach [68]. It employs the alternating variable method (AVM) local search to generate test data for procedural programs. Since then, the use of search based techniques has been widely investigated and shown great potential in software testing [68]. According to a recent survey, there has been a rapid increase in the number of papers in the area of SBST since 2001 [52]. The most widely used search based techniques are genetic algorithms (GAs), genetic programming (GA), and hill climbing (HC) [52].

In 2004, Tonella [97] applied a genetic algorithm (GA) testing technique to test object-oriented programs such as Java classes and developed a tool called eToc. In this approach, a population of test cases was generated when a new branch was targeted. Test cases were well formed and represented an actual execution of sequence method calls

with their caller and inputs. As a result, special mutation operators and the crossover were implemented to enforce the feasibility of the new generated test cases. The fitness function used to guide the search was the ratio of the number of control dependences traversed during the execution of a test case over all the control dependences that lead to the target branch. eToc was evaluated on seven different Java classes and covered relatively high branches when compared to random testing [97].

Despite the fact that eToc was the first search based technique for testing object-oriented programs, eToc does not address several object-oriented features (e.g., encapsulation, and inheritance). Furthermore, eToc's fitness function does not exploit the branch distance [68]. McMinn [68] argues that if the fitness function uses only the number of control dependences, the search will have no guidance on how to enter nested branches and cover the target branch. In other words, the search space will contain plateaus, which degenerate the search to a random search if it reaches such a plateau.

The predominant approach in SBST is to separately target each test goal (e.g., a target branch), and then a separate search is undertaken to cover this goal (e.g., [57][97][101]). One of the problems facing this approach within evolutionary algorithms (e.g., GA) is that when each branch is individually chosen, the predicate of that branch might not be executed by any of the test cases in the population [68]. As a result, the following studies addressed the issue of targeting a single test goal each time and proposed approaches in which all test goals (e.g., target branches) are targeted simultaneously [14][16][31].

Arcuri and Yao [14] applied and analyzed different search algorithms on the testing of Java container classes (e.g., Vector, Stack, LinkedList, and Hashtable). Hill climbing (HC) with random restarts, a genetic algorithm (GA) and memetic algorithm (MA) were used and compared. Their empirical study showed that the MA results were the

best among the algorithms. Arcuri and Yao [14] also proposed a more advanced fitness function that integrates the normalized branch distance with the total number of covered branches and the length of test cases. Based on the empirical evaluations, similar to eToc [97], Arcuri and Yao [14] approach only targets individual classes (i.e., container classes), and cannot generate sequence of method calls that involve more than one class [94].

Barsei et al. [16] proposed a hybrid global-local search (MA) tool for Java classes called TestFul. Their approach combines GA and HC to generate tests that cover the maximum number of branches of the class under test (CUT). The GA is used to search for the test that has higher coverage and explore all the internal states of the CUT. The HC is used to target uncovered branches. Barsei et al. [16] use a fitness function that consists of three parts: the length of the test case that must be minimized, and both the number of covered statements and branches that must be maximized. TestFul is a semi-automated approach, 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 that can be used as concrete implementations of the abstract classes and interfaces [16]. As a result, the empirical evaluation of TestFul was only on 15 classes [16][76]. However, TestFul generated tests with higher branch coverage when compared to eToc [16].

EvoSuite [31] automatically generates and optimizes whole test suites towards satisfying a coverage criterion, e.g. branch coverage. EvoSuite uses a GA to evolve and optimize whole test suites that consist of a different number of individual test cases to alleviate the problem that derives from the infeasibility and difficulty of targeting each branch in turn. In addition, the search is guided by a fitness function that calculates the number of executed methods and the minimum normalized branch distance for each of the uncovered branches in the CUT. The empirical study showed that EvoSuite is

superior to the traditional approach that targets one branch at a time [31]. Recently, the GA search in EvoSuite has been combined with the local search alternating variable method (AVM) to optimize the values in a specific test case of a test suite [33]. Their result showed that the combined techniques increased the branch coverage by up 32% over GA [33].

While our work is broadly equivalent, it is not identical to the algorithms used by Arcuri and Yao [14] and Barsei et al. [16]. Our approach is different in three aspects. First, our approach is fully automated to deal with different object-oriented features, such as inheritance and polymorphism. Second, we fully statically analyze the CUT and automatically leverage method dependency relations (MDR) [105] to avoid exploring the whole search space and to augment both the efficiency and effectiveness of the search. Lastly, we implement a new strategy to seed different types of values (i.e., primitive, non-primitive and arrays) during the search process to generate test data inputs that cover complex program branches.

3.2 The Chaining Approach and State Problem

It has been shown that the internal state of objects can cause problems for search based software approaches [52]. As defined by Harman [52], the internal state problem occurs when certain target branches are controlled by predicates whose values depend on state variables. As a consequence, the search requires finding a sequence of method calls, which involves creating objects and invoking methods from multiple classes, to change the state variables into the proper values required to cover the target branches [94]. These types of optimizations can lead to a loss of guidance in fitness function [72].

McMinn and Holcombe [72] argue that in the presence of the state problem, an

arbitrary call to the method under test (MUT) may not cover a target branch. Therefore, extra guidance must be provided to search for method calls and data inputs that change the object state to the right internal state [72]. As a result, McMinn and Holcombe proposed a solution using the chaining approach developed by Ferguson and Korel [27].

The chaining approach [27] originally was used to generate test data for programs and applied data flow analysis to find previous program statements on which a target branch depended. Those statements are sequences of events, and the alternating variable method (AVM) search was used to generate data input to execute these statements to cover the target branch. However, McMinn and Holcombe [72] combined the evolutionary search with the chaining approach for testing procedural programs. In their approach, if the search failed to cover a target branch, the chaining approach performed and identified a relevant sequence of functions that needed to be executed to cover a target branch. A simple initial experiment with a preliminary version of the system was conducted, and the results showed the effectiveness of the proposed approach.

Although applying the chaining approach can be an effective way to generate test data, evaluating all possible transitional statements to cover the target branch is a very expensive process [52]. As a result, the aforementioned approaches were only evaluated with a small number programs. In contrast, our approach is fully automated and handles different features of object-oriented programs and their size, which can vary by an order of magnitude [1].

3.3 Input Domain Reduction

The input domain reduction technique typically deals with the identification of irrelevant variable inputs and eliminates them from the search space and the scope of testing to

reduce test effort [48]. Harman et al. [48] were the first to theoretically and empirically explore the input domain reduction for the SBST.

The search space is based on the set of possible input domains of the program under test [48] . In particular, the input domain in object-oriented program testing is generally considered all the public methods and constructors of the class under test (CUT), including the primitive and non-primitive parameters [48]. The search-based approaches for object-oriented programs are hindered by the explosive size of the input domain of the object-oriented programs [94]. It is possible, however, that some methods in object-oriented programs do not determine whether a branch will be covered or not [50]. The input domain reduction avoids the generation of irrelevant methods and variable inputs from the input domain of the CUT using various techniques. Thereby, reducing the size of the search space and potentially enhancing the search process [84].

In the next subsections, we provide an overview of three techniques that have been used to perform input domain reduction for effectively improving code coverage results. These techniques include program slicing [102], purity analysis [87], and method dependence relations (MDR) [105].

3.3.1 Program Slicing

Program slicing is a static technique that focuses on selected aspects of semantics based on a slicing criterion that helps to create a reduced version of a program [102]. The slicing criterion captures the semantic of interest, while the process of slicing deletes any parts of the program that cannot affect the slicing criterion [102]. Based on the slice criterion, it is possible to construct two forms: a backward or a forward slice. A backward slice contains all the statements of the program that could influence the slicing

criterion, whereas a forward slice contains those statements which can be influenced by the slicing criterion [49].

The program slicing technique can therefore provide precise guidance for the search process since it slices away irrelevant statements of the program under test and helps identify which input variables can not influence the coverage of a target branch. As a result, the program slicing technique has been used to reduce the input domain of search-based software testing (SBST) approaches [49][48][50][71].

Harman et al. [48] were the first to theoretically and 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 [48].

In a separate study, Binkley and Harman [18] conducted a simple experiment to show how the analysis of predicate 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 the analysis of predicate dependency with the optimized search required fewer fitness evaluations.

Harman et al. [50] 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 [102]. 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. Then, an evolutionary testing approach is conducted

only on the identified relevant parameters. However, a test target branch does not necessarily depend on any parameters of the method under test, and might only depend on a member field. Despite the fact that defined member fields were not considered in their domain reduction, their results showed a decrease in test effort with reduction and, most importantly, an increase in the number of branches covered.

3.3.2 Purity Analysis

Salcianu and Rinard [86][87] described a systematic purity analysis for Java programs. Their analysis classifies a method as pure if it does not modify objects that existed before the execution of that method [87]. More interestingly, their analysis can also recognize *read-only* and *safe* parameters when the method is impure.

- A parameter is *read-only* if the method does not mutate any object reachable from the parameter [87].
- A parameter is *safe* if it is *read-only* and the method does not create any externally visible heap paths to objects reachable from the parameter [87].

These definitions are useful in the context of search-based test case generation, as it provides a way to automatically identify and remove irrelevant methods that have no affect on whether a branch will be covered [46] [84].

Arcuri and Yao [13] proposed a technique called Dynamic Search Space Reduction (DSSR) that can be applied to any type of object-oriented program. 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: container classes, which are a typical benchmark in testing of object-oriented programs

[29]. 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 *read-only* or not. The empirical results showed that DSSR usage improved the efficiency of the search algorithms, particularly the hill climbing local search, in terms of speed and number of steps to reach a global optimum. However, the applicability of DSSR to non-containers was unclear. Some studies have suggested that containers have quite different behavior than more general code [43].

Barsei et al. [16] 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 of the CUT. A method can be: (1) a mutator, when it may change the objects state; (2) a worker, when it does not change the objects state but it may perform some computations, or (3) an observer, when it does not change the objects state and does not perform any additional computation [17]. TestFul exploits the information and prunes methods from the test case that have no impact on the targeted branch before starting the HC search.

EvoTest [89] and eCrash [84] approaches leverage purity analysis to reduce the input space of object-oriented programs. 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 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 in-

clusion 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.

3.3.3 Method Dependence Relations (MDR)

In contrast to aforementioned approaches, our approach statically analyzes the source code of the CUT to precisely identify only those member fields (primitive and non-primitive) or parameters of the MUT that would be relevant to covering uncovered branches. Then, it leverages MDR to automatically guide the search to generate a sequence of method calls that produces the desired values for member fields or parameters that have impact on target branches. Zhang et al. [105] 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 and ; reads field and writes it, it is declared that has write-read dependence relation on g.
- *read-read relation*: If methods and both read the same field , 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) [104].

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

MDR is useful for testing write-read related methods, as it has a high chance of exploring new program behaviors and states [105]. In addition, it is especially useful in the context of search-based software testing approaches, as the search domain of object-oriented programs can be reduced by automatically identifying and eliminating read-read related methods. In addition, MDR can also identify candidate methods that modify a specific member class field [13].

Regarding the reduction of the search space based on member fields, we are aware of the work of Thummalapenta et al. [94]. In that work, the Seeker tool combines both dynamic symbolic execution (DSE) and static analysis. Their approach statically analyzes an inter-procedural execution trace which gathered during the execution of DSE to identify a target member field that needs to be modified for covering a target branch. Then, their approach exploits method-call graphs to identify methods that modify the target member filed. However, static analysis used in our work differs completely from the static analysis used in their approach. Their approach uses method-call graphs while the work we proposed uses MDR [42].

3.4 Seeding

When a class under test (CUT) contains branches that depend on particular constant values, randomly generating the right values can be very challenging. However, covering

such branches can be easier if the constant values are extracted from the CUT and used instead of randomly generated values [28].

Many works proposed different seeding strategies to enhance the efficiency and effectiveness of SBST. For example, Alshraideh and Bottaci [5] proposed a seeding strategy in which string constants are extracted from the source code of the CUT and used to seed the initial population of the GA. Their empirical study showed using extracted string constants can improve the branch coverage of the CUT. McMinn et al. [74] also proposed an approach for generating test cases involving string inputs, where the potential string values are extracted from web queries. The empirical results show that their approach improves branch coverage for 15 of the 20 Java classes. Recently, Alshahwan and Harman [4] introduced a seeding strategy, Dynamically Mined Value (DMV), in testing PHP web applications using SBST. Their approach dynamically collects constant values from the returned HTML and associates them with their respective input fields. Then these values are randomly seeded in the search process when targeting their associated branches. Their results show the DMV seeding approach significantly increases the efficiency and effectiveness for web application testing.

Fraser and Arcuri [28] conducted a study on 20 Java projects using EvoSuite, concluding that the use of seeding can strongly improve performance of an evolutionary search algorithm. During the search process, with a defined probability an extracted constant value, as opposed to a randomly generated new value, is used. The results showed best coverage was obtained using a 20% probability of extracted constants on all the case study. However, Similar to the aforementioned works, their strategy seeding considered only primitive constant types (e.g., numbers, strings). Recently, Sakti et al. [85] also proposed another strategy seeding in which both the primitive constant values and the null constants were extracted from the source code of the CUT. Their seeding

approach defines a seeding probability for each extracted constant based on the number of occurrences of the constants in the CUT. The approach was implemented to seed a random testing (RT).

In object-oriented programs, however, branches are involved with both primitive constant values, and also with non-primitive values, such as instances of classes, and arrays. For example, assume that a search technique test generation tries to cover branch B1 at line 3 in the MUT foo as follows:

```
1.  public boolean foo equals(Object obj) {
2.        If(obj instanceof Foo)
3.            return true; //B1
4.        Return false
4.  }
```

Figure 3.1: foo class

The signature of the MUT required an instance of Object (line 1). In this case, the search technique will randomly try to choose a concrete value for value obj from all the classes assignable to Object. The chances of selecting a class foo for value obj can be very small because the branch using **instanceof** operator (line 2) which offers no guidance to the search.

Our seeding approach, therefore, differs from the previous works in two aspects. First, beside the primitive constant values, our approach extracted other available information from the predicates of the target branches in the CUT, such as type of classes, null constant values, and arrays indices. Second, our seeding strategy with a defined probability uses only the proper values for the uncovered branches from which they were collected.

Chapter 4: Improving Genetic Algorithm via Method Dependency Relations

4.1 Introduction

Significant success has been achieved by applying search-based software testing (SBST) to the problem of automated structural test data generation [13][16][31][50]. SBST formulates the process of generating test inputs as a search problem, and employs meta-heuristic techniques (e.g., evolutionary algorithms) to find test inputs [68]. The power of SBST lies in the use of fitness functions which provides effective guidance towards the optimum solution. Thus, the effectiveness of the search algorithms is improved as long as the fitness function distinguishes between better and worse solutions [13].

Many different search-based testing techniques for object-oriented programs have been proposed to automatically generate sequences of object constructor and method calls such as *TestFul* [16], *EvoSuite*[31], or *eToc* [97]. A major issue with most of the existing approaches is that they consider the whole search space of possible input values and method calls to the program under test. In practice, generating method calls to generate desirable objects has been a significant challenge for automated test input generation approaches, partly because of the huge search space of possible method calls [59]. This problem has been significantly highlighted in several research studies [34][50][94][105].

This chapter introduces an automatic approach, called GAMDR[1], to effectively reduce the size of the search space and generate desirable object constructor and method

[1]The name is derived from improving Genetic Algorithm via Method Dependency Relations

calls. The purpose of our approach is not to exhaustively explore the search space, but to improve branch coverage by intelligently guiding the search toward finding regions in the search space with method calls that produce desirable object instances. GAMDR uses a similar GA algorithm used for the empirical work in [15] and [16] and augments it with static analysis techniques adapted from recent work on random testing [105][93].

Previous works on SBST to reduce the search space for object-oriented programs has tended to be evaluated on relatively small scale systems [13][16][84]. However, GAMDR can handle a large scale programs of Java as well as a number of open source programs (See Section 4.4). An empirical study was performed in which GAMDR was compared with a well known evolutionary testing tool EvoSuite [31]. In order to establish a fair comparison the GAMDR was also compared with two other different approaches: the basic implementation GA without MDR enabled [15][16], and pure random testing RT [22]. The goal of the empirical study was to examine how MDR impacts the effectiveness and efficiency of the GA effort in terms of improved branch coverage over a limited time period.

The major contributions of this chapter are as follows:

1. A fully automated search-based testing approach for Java that addresses the challenging problem of SBST.

2. A description of how MDR is incorporated into the design and implementation of the GAMDR approach.

3. An empirical study that evaluates the effectiveness and efficiency of the GAMDR approach.

The rest of this chapter is divided into the following: Section 4.2 illustrates some of the problems that search-based approaches face. Section 4.3 presents our proposed

approach for improving branch code coverage. We explain how GAMDR leverages data dependency and method dependency relations to guide the search process in improving code coverage and also cover its implementation id details. In section 4.4 presents the empirical study used to evaluate GAMDR alongside the research questions answered, and it also represents threats to validity. Section 4.5 concludes the chapter.

4.2 Motivation

In this section, we illustrate some of the problems that we empirically observed by applying the state-of-the-art search based tool, called EvoSuite [31], through illustrative examples taken from the Commons-lang3[2] and NanoXML[3] projects. These examples are intended to illustrate how the size of domain input and the presence of complex data dependencies are a key limiting factor of achieving high branch coverage for SBST.

4.2.1 Input Domain Size

The size of the search space of potential data inputs is a key factor affecting the effectiveness of any search-based approach [71][48]. To illustrate this issue, consider the `ArrayUtils` class from `Commons-lang3` example shown 4.1. The class contains 229 different public methods to test, each of which takes primitive and/or array arguments. Although the class contains 1104 target branches, most of these branches are trivial and not difficult to cover. For example, supposing that the method under test named as MUT is `subarray` (lines 3-17), and it takes three different arguments. The first argument is an array of characters, followed by two integer arguments used as the start and

[2]org.apache.commons.lang3
[3]http://nanoxml.sourceforge.net/orig/

```
1.   public class ArrayUtils {
2.   ...227 more public methods ...
3.    public static char[] subarray(final
     char[] array, int startIndexInclusive, int
     endIndexExclusive) {
4.         if (array == null) {
5.              return null; \\B1
6.         }
7.         if (startIndexInclusive < 0)
8.              startIndexInclusive = 0;
9.        if (endIndexExclusive >array.length)
10.             endIndexExclusive = array.length;
11.     final int newSize = endIndexExclusive -
     startIndexInclusive;
12.        if (newSize <0)
13.             return EMPTY_CHAR_ARRAY;
14.     final char[] subarray = new char[newSize];
15.     System.arraycopy(array, startIndexInclusive,
     subarray, 0, newSize);
16.    return subarray;
17.    }
18.  }
```

Figure 4.1: ArrayUtils class

the end indices. The method returns a new array containing the elements between the start and the end indices. Let us assume that the true branch of the predicate "if (array == null)"(at line 4) is the target branch B1.

In practice, the test data generation process (i.e., search process) explores various combinations of method calls and the input data for all their parameters [50]. The number of possible method calls of the target class (i.e., `ArrayUtils`) is very large and the possible input data values of each method increases the search space even further. As a result, the search process is quite difficult and time consuming which tends to lower

the effectiveness and efficiency of the search in general.

The relevant method calls and the parameters which affect the coverage of a target branch constitute only a small portion of the entire search space. We can observe that, the only parameter of the target method that can influence the coverage B1 is the first parameter, which is the `array` parameter, and it is required to hold `null` value. As a result, the search process should explore the search space for these relevant method calls and parameters instead of investing time on the entire search space.

As revealed by our experimental results, pure random testing (RT) achieves 99% versus EvoSuites 68% branch coverage of the `ArrayUtils` class. We speculate the low branch coverage of the EvoSuite because it failed to call the more relevant methods and their parameters to cover certain target branches due to the extreme number of the public methods of the `ArrayUtils` class. In other words, EvoSuite should only consider and choose these relevant methods and parameters during the search process to cover certain target branches and satisfy all-branch coverage criteria [50].

4.2.2 Control and Data Dependencies

Search-based test data generation also encounters extra challenges in generating data inputs in the presence of complex predicates of the method under test (i.e., MUT) [72]. A major challenge arising from the presence of complex predicates is the search has to navigate all the entire search space to generate a desirable sequence of method calls. Thus, the search process may require very specific guidance to find the valid sequence of method calls and the input values of their parameters.

To understand the effect of complex predicates, consider the source code of the classes `CDATAReader` and `StdXMLReader`, which are part from `NanoXML`, given in Figure 4.2.

```
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(char[] buffer, int offset, int
     size)throws ...  {
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.               char ch3 = this.reader.read(); \\B4
22.      ...more if statements ...
23.      }
24.    }
25.   }\\end of method public int read line 10
26.    ...3 more methods ...
27.  }\\end of class CDATAReader


28.  public class StdXMLReader implements IXMLReader {
29.     ...
30.     public StdXMLReader(String publicID, String systemID)
     {
31.       URL systemIDasURL = null;
32.       ...
33.     }\\end of public StdXMLReader
34.     public static IXMLReader stringReader(String str) {
35.       return new StdXMLReader(new StringReader(str));
36.     }\\end of public static IXMLReader stringReader
37.     ...20 more methods ...
38.  }\\end of class StdXMLReader
```

Figure 4.2: Two classes taken from the NanoXML project

We consider the method `read` (lines 10-24) as the target method, which reads data from another reader until the end, and returns the number of characters read, or -1 if at EOF. The method `read` takes as inputs an array of character and two integer variables.

A `CDATAReader` test must satisfy both values and order method calls to achieve full or at least high branch coverage. We can observe that, creating a `CDATAReader` object requires a valid `StdXMLReader` (line 28) object, which is a concrete implementation of the interface class `IXMLReader`. As a result, a valid sequence of method calls requires calling methods in a correct order to create the desired objects: a valid `IXMLReader` object must be created before a `CDATAReader` object.

We can also observe that, the target method (lines 10-24) contains some branches that require a particular character value, such as ']', at line 18. In fact, some branches'predicates involve a Boolean value, such as target branch B3 at line 19, i.e. the flag problem [68]. 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 [68].

Suppose the target branch is the true branch of the predicate "if (ch2 == ']')"at line 21, which is only executed when the non-primitive filed `reader` contains ''']]''. If the search process fails to find and generate method calls and parameters that satisfy these constraints, the generated tests will fail cover the target branch B4. For example, the constructor of `CDATAReader` needs to have a valid `IXMLReader` object with all its field members correctly initialized as a parameter.

Despite the fact that the class `CDATAReader` contains only 4 public methods, our experiment revealed that RT, simple GA, and EvoSuite [31] could only achieve 66%, 71%, and 68% branch coverage of the `CDATAReader` class, respectively. As shown

through this example, it is clear that EvoSuite faced a challenge to achieve high branch coverage of the class `CDATAReader` due to the requirement of complex method calls. This is because there is no guidance encoded in the fitness function identifying which constructors, methods or parameters must be called in order to cover certain target branches.

This result also supports the belief that the applicability of the search-based test data generation techniques are limited not only when the search space is large but also when it does not take into account data dependencies within the CUT [14][73].

4.3 GAMDR

GAMDR is a SBST approach which uses a combination of genetic algorithm (GA) and static analysis technique (MDR). It can be used to generate test data for a given Java class which achieve high branch coverage for that class. During the search process toward finding test data inputs, GAMDR does not attempt to cover each target branch individually, but it tries to cover all the target branches at the same time. This implementation is likely to accelerate the search towards the global optimum (i.e., total branch coverage) because it does not waste efforts and time on infeasible target branches [31]. In addition, the search is guided by a fitness function that was introduced by Fraser and Arcuri [31]. In order to calculate the fitness function, the CUT must be instrumented.

As shown in previous sections, the search-based approaches EvoSuite and pure GA when executed against large classes, such as ArrayUtils class, in limited time often end up with only small regions of the search explored and fail to achieve high branch coverage of the classes due to the large size of the search space. These approaches also face a challenge of achieving high branch coverage when executed against classes, such as

CDATAReader class, that require of the generation of a sequence of complex method calls.

To cope with such challenges, GAMDR leverages MDR to direct GA toward effectively and efficiently exploring the search space with the following steps. Given a class to test, GAMDR first statically analyzes each target branch predicate and performs data dependency analysis to identify the relevant member fields (i.e, attributes) and/or parameters that affect execution of the target branch. After all relevant member fields and parameters have been identified, GAMDR uses MDR to identify all the methods and/or constructors of classes that change (i.e., write) the value of those identified relevant member fields and parameters. Finally, GA specifically narrows down the search space and only explores these identified relevant methods and constructors to create a sequence method calls that are required in order to cover certain target branches. Combining GA with MDR has a number of advantages:

1. It focuses on the root cause of getting the branch targets to be covered.
2. It focuses only on the relevant parts if the test input that affect the execution of the target branches.
3. It implements a domain reduction mechanism to improve the search space exploration.

Unlike previous search-based approaches, these strengths together enable the proposed approach to steer the search in a directed, automated manner toward effectively and efficiently exploring promising areas of the search space. This approach allows the search to explore high complexity code in order to achieve high branch coverage.

4.3.1 Example

This sections explains the search mechanism in GAMDR using the same illustrative example shown in Figure 4.2, and assumes that GAMDR tries to cover B4 at line 21. It can be observed that the non-primitive field `reader` affects whether the target branch B4 is covered. Another observation is that, the `reader` object, which passes as an argument of the `CDATAReader`'s constructor at line 5, should already exist and precede the call of `CDATAReader`'s constructor.

Based on these observations, GAMDR applies MDR to identify both irrelevant member fields of the target class and parameters of the target method. Then it excludes the irrelevant member fields and parameters from the scope of the search process in order to improve search efforts.

The resulting MDR analysis identifies only relevant methods and constructors that can influence the coverage of the target branch B4. For example, MDR identifies the constructor of `CDATAReader` that writes the field `reader`. Then, MDR also identifies both the class constructor of `StdXMLReader` at line 30 and the method `stringReader` at line 34, because they both return objects that can be used to replace the interface class type argument in the `CDATAReader` constructor. As a result, GA tries to generate test data and method calls for these relevant methods and constructors instead of investing time on all constructors, methods and parameters.

This combined MDR and target branch predicate extraction information allows GAMDR to generate more effective sequences of method calls. These generated sequences can cover branches that require complex method calls. Our results show that GAMDR achieves 90% branch coverage of the `CDATAReader`, which is 23% higher than pure random testing RT, 22% higher than EvoSuite and 19% higher than a simple GA.

4.3.2 Approach

In this section, the concepts of the GAMDR approach are presented. As depicted in Figure 4.3, GAMDR consists of three different components.

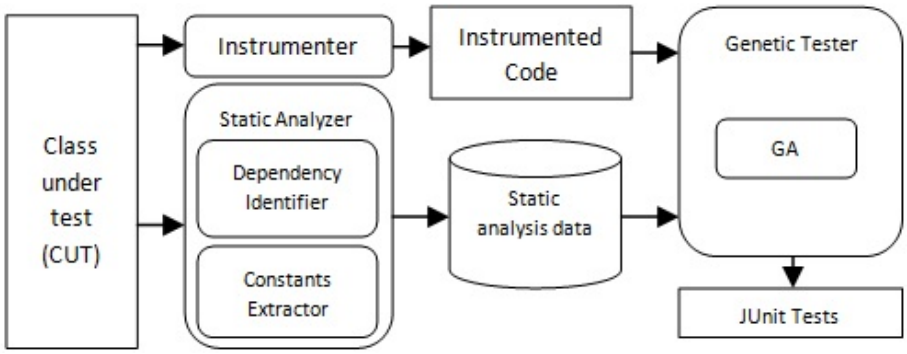


Figure 4.3: Overview of GAMDR

- **Instrumenter:** The original source code of the CUT is instrumented at byte-code level to measure the coverage values and calculate fitness function. In our experiments, we used Soot [40] for analyzing and instrumenting Java byte-code.

- **Static Analyzer:** The static analyzer component is used to identify method dependency relations (MDR) based on the set of the fields that may be read or written by each method [105] and collect specific constant values (primitive and non primitive) from predicates. The results are stored in a repository and used later by GA search, which helps in generating required values and sequences of method calls that explore more branches and increase code coverage of the CUT.

- **Genetic Tester:** This component conducts evolutionary testing using GA, which is derived from [14][16] but additionally implements MDR. The GA targets all

the branches of the CUT at the same time, and it iteratively manipulates the population by applying fitness function, selection, crossover, and mutation in order to eventually cover all the target branches.

We next present more details on two key components in GAMDR: The Static Analyzer and the Genetic Tester components.

4.3.2.1 Static Analyzer Component

Using the existing GAs approaches such as [31][16], we can see that during the search process some methods or parameters are randomly changed into the existing potential solutions (i.e., test cases). However, certain branches are required for more favorable changes [34]. In addition, when the branch distance does not offer enough guidance to automatically construct complex objects through a method sequence of calls, the search with a GA will degenerate to a random search [81]. When this is the case, the search tends to stagnate and fail to find optimal solutions.

The key idea behind our approach is in using a static analysis to identify relevant methods and constant values for each target branch, and then use them during the search process (e.g., mutation operation). Thus, our proposed solution more effectively improve the chance of selections of most relevant methods and parameters that may help to cover certain branches.

Our Static Analyzer Component (SAC) takes the advantages of MDR [105] and is able to capture essential information (e.g., constructors, methods, and parameters) from the source code. The information then is used to assist the search in two major ways. First, it guides the search to those parts of the existing solutions in the population where changes are most likely to affect coverage of more branches. Second, it identifies

irrelevant methods and parameters, excluding them from the scope of testing in order to reduce test efforts.

The novelty of our approach is that SAC helps the search of the GA to avoid premature convergence and to increase the speed of exploring the search space. To extract relevant information from the class under test (CUT), we use static analysis and implement this phase in two stages: **Dependency Identifier** and **Constants Extractor**

- **Dependency Identifier.** In the dependency identifier stage, we analyze the CUT and start by identifying the constructors and methods of all target branches. For each target branch, we perform backward analysis, and precisely identify whether a parameter of the method contains the target branch (named as a target method) or if a member field of a class can help to cover the target branch. Knowing which member fields of the CUT are affecting the target branch, we can greatly improve GA performance by calling only methods that modify those fields, rather than the entire methods. As a result, for each member field, we use MDR to identify the methods that modify the member field (i.e., `write-read relation`). In addition, if a parameter of the target method affects the coverage of the target branch, we identify all the methods that write in the target method (i.e., `read-write relation`). However, if the identified parameter is not a primitive type, we identify the methods return the same type object that can be passed as an argument to the target method (i.e., `accessed-data relation`).

- **Constants Extractor.** During the constants extractor stage, if a target branch involves primitive constants or strings, the extract constants component is extracting constant values from each target branch. These values are seeded during the search of the GA to help cover their associated target branches. However, in some

cases a target branch is dependent on non-primitive constant type, and the signature of the target method does not reveal what the exact type of the parameter should be [32]. In fact, this problem is very common in Java and is known as Java Generics. As a result, beside the primitive constant values, constants extractor extracts other available information from the predicates of the target branches in the CUT, such as type of classes, null constant values, and arrays indices.

4.3.2.2 Genetic Tester Component

For an algorithm to be considered genetic, we need to define a representation of test cases as individuals, a fitness function, and the genetic operators: selection, crossover and mutation. The selection operator is used to select the prospective parents, the crossover operator is used to create new individuals by mating parents, and the mutation operator is used to mutate the individuals. There is a large body of literature regarding the role and effects of the various fitness functions, crossover operators, and mutation operators [10][58][70].

In Genetic Algorithms, mutation operator plays an essential role, which slightly modifies the individuals, i.e. test cases, with a relatively small probability. In general, the mutation operator is used to introduce new information (e.g., a method call) into the population and is considered a random variation [58]. Mutation operators (e.g., modifying input primitive values or inserting/removing method calls) are randomly performed to preserve diversity of populations, and avoid the search to trap in a local optimum [58].

Nevertheless, whenever mutation occurs, the chance of choosing the right method calls or primitive values (where changes are most beneficial) is very low. Thus, such a

"blind" mutation has two different problems. First, insufficient guidance to the required data inputs can cause unnecessary computation expense [34]. This is due to an inability to explore promising areas in the search space. Secondly, randomly flipping methods or manipulating an input primitive value may fail to generate high quality new solutions [13]. This can lead to increased chances of premature convergence (i.e., trap in a local optimum) due to the lack of diversity in the population [71].

GAMDR thus exploits MDR to narrow down search space and direct mutation operators to the most beneficial regions in the search space that lead the global optimum, or high branch coverage of the CUT.

We have opted the following individual representation, fitness function, and the selection, crossover, and mutation operators:

1. **Individual representation**: An individual can be viewed as a sequence of method and constructor calls, we decided to use an individual representation similar to [8, 13], because it is easy to apply and manipulate. GAMDR is not only restricted to primitive datatypes, but also can handle arrays of any type. As a result, each individual consists of a set of statements that are a constructor, method call, or array input:

    a) ***Constructor statement***: represent 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(charArray_0,10,20);*. Parameters of constructors and method classes can be randomly generated and initialized depends on their types.

    c) ***Array statement***: directly accesses a *public* field and changes its value of

an object, e.g., *int CDATAReader.field = 0;* {*'A','l','i'*};.

d) ***Field statement***: directly accesses a *public* field of an object and changes its value , e.g., *CDATAReader.field = 0;*.

For a given class to test, the test cluster [100] is automatically defined. The test cluster of the class under test includes all the public constructors of the available classes, public methods, and fields, and 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 test cluster. In addition, returned non primitive objects are stored in a pool and served as a target object or parameter object for succeeding statement calls.

2. **Fitness Function**: GAMDR uses a fitness function to determine if an individual is to be selected for reproducing in the subsequent generations. We use the fitness function in equation (1) to guide the search and it is similar to the one proposed by [13][31]. The fitness function uses branch distance (BD) and keeps track of how close an individual is to cover all reachable branches, but not executed yet.

$$f(i) = \sum_{b_j \in B} BD(b_j, i), \text{ where B is the target branches of the CUT} \qquad (1)$$

And, we use equation (2) for measuring BD.

$$BD(b_j, i) = \begin{cases} 0 & \text{if brnach j is covered} \\ k & \text{if brnach j is reached} \\ 1 & \text{otherwise} \end{cases} \qquad (2)$$

Where k is a normalizing function and its value within [0,1], and we use the normal-

ization function [8]: $k = \frac{x}{x+1}$. The function $BD(b_j, i)$ shows how close an individual i is to cover that not covered branch j. For instance, for predicate $a < 10$ when the value of a is 2, then the distance to the false branch is $x = 10 - 2$ [34].

While BD often gives good results, it can deceive the search and lead to longer individuals without increasing the coverage, which is called *bloat* [30]. Therefore, when during the search process there are two individuals have the same coverage, we always prefer the shortest individual.

3. **Genetic Operations**: GAMDR implements common genetic operators, i.e., selection, crossover, and mutation, to manipulate and evolve successive populations. Following is a summary of these operators:

   a) ***Selection***: In this operation two parents are selected for reproductions (i.e., crossover). We implement tournament selection [68]. In this selection mechanism, two individuals are randomly selected. Then, a random number is generated. Finally, we select the fitter individual if is less than r, otherwise the less fit individual is selected. However, if two individuals have the same fitness values, the shortest individual is selected to prevent *bloat* [30].

   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. Rather than just randomly change statements of the chosen individuals, GAMDR uses MDR to direct mutations towards relevant statements where changes may

help to result better fitter individuals and increase exploring the search space. Therefore, GAMDR randomly chooses a reached but not covered branch and analyzes its predicates. Then, GAMDR precisely identifies the relevant type of elements that are involved in executing of the branch, e.g. member field, parameter method, or/and constant values. Consequently, GAMDR directs the mutation operators to explore those identified relevant statements (i.e, constructors, methods, and parameters). Finally, for a chosen individual with a length n, GAMDR randomly apply one of the following operations with probability 1/3.

- **Remove**: All irrelevant statements are removed, as well as a chosen statement from the identified related statements is removed from the chosen individual with a probability r.
- **Insert**: Insert a random number r, where $1 \leq r \leq n$, of identified relevant statements in a random position in the chosen individual.
- **Change**: Each identified relevant statement and parameter is changed in the chosen individual with probability r.

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.

4.4 Empirical Study

In previous sections, we have presented the GAMDR approach for automated structural coverage for Java programs. Our approach uses genetic algorithm (GA) to perform test data generation, and utilizes the method dependency relations (MDR) to guide the

search toward efficiently and effectively exploring the search space.

In this section, we present an evaluation of the effectiveness of GAMDR in improving branch coverage testing. In order to extend the applicability of the study, we compared GAMDR's effectiveness against three different approaches: a simple GA, pure random testing (RT) [22], and the EvoSuite [31]. We assessed the effectiveness of MDR by empirically comparing the performance of the search algorithm GA without MDR, to the performance of the algorithm with MDR. We refer to these algorithms as GA and GAMDR respectively. We also compared GAMDR with RT in order to make sure that the success of GAMDR is not due to the simplicity of the test subjects [3]. In addition, to allow for a fair comparison, we compared GAMDR with RT that implemented in generating the initial population of GAMDR. We also compared GAMDR with EvoSuite with two reasons. First, EvoSuite is a mature tool that applies a genetic algorithm to generate test data that achieve high code coverage for Java classes. Second, EvoSuite is publicly available and has been successfully run on a variety of Java projects [10][28][30].

Although we explored the possibility of comparing *Seeker* [94] to GAMDR, but we could not perform such comparison because *Seeker* targets .Net programs, particularly C#, whereas GAMDR targets Java programs. *TestFul* [16] also could not be applied in our evaluation because the tool is semi-automatic and requires the user to provide some XML description of the class under test (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 (see section 4.1.1) that we used in our experiments makes it difficult to compare GAMDR with *TestFul*.

The goal of the empirical study is to investigate the effectiveness and efficiency of GAMDR when compared with representative test generation tools. The research ques-

tions to be answered by this study are therefore as follows:

RQ 1: How effective is GAMDR in achieving branch coverage in comparison with representative test generation approaches? This research question focuses on determining the influence of the MDR approach for improving branch coverage. To do so, we compared GAMDR with a simple GA (i.e., without MDR enabled), RT and EvoSuite. We also compared their results in terms of the achieved code coverage.

RQ 2: Given a fixed search time, can MDR improve the efficiency of a simple GA in achieving branch coverage in comparison with representative test generation approaches? This research question aims to see if MDR has the potential to improve the efficiency of the simple GA in exploring of a large search space and finding the required test data. To address this research question, we ran GA, GAMDR, RT, and EvoSuite on each target subject with a short time limit. The more quickly additional branches are covered, the more efficient the test generation approach.

RQ 3: What types of branches does GAMDR fail to cover? The aim of the third research question is to identify the challenges remaining for our future research. To this end, we selected classes where GAMDR achieved low coverage, and manually analyzed and investigated the reasons for low coverage.

The research questions were addressed using various open source Java projects, as described in the next section.

4.4.1 Test Subjects

In the empirical study, we used five popular open source projects as test subjects in our evaluations. Table 4.1 shows some of their characteristics such as the name, the number of classes, methods, NCSS (i.e., Non Comment Source Statement) lines of code, and branches for each subject. Classes, methods, and NCSS were measured at the byte-code level, using the Javancss[4] tool. However, the number of branches were measured at the byte-code level using GAMDR .

Test Subject	#Classes	#Methods	NCSS	#Branches
Commons Codec	41	654	3,269	1,373
Commons CLI	11	126	677	288
Conzilla	13	67	377	120
jdom2	40	647	3,196	978
lang3	55	1,558	9,182	5052
NanoXML	26	322	1,984	571
Joda-Time	57	1,489	9,152	2,207
Total	**243**	**4,863**	**27,837**	**10,589**

Table 4.1: Details of the test subjects used in the empirical study.

In total 175 classes were used, and were selected because they were widely used as benchmarks for evaluating different SBST approaches, such as EvoSuite [31] and Delver [74], and non SBST approaches, such as JTExpert [85] and Palus [105]. Apache Commons Codec (Commons Codec) is an implementation of common encoders and decoders such as Base64, Hex and URLs. Commons CLI is a library that provides an API for parsing command line options passed to programs. Conzilla is a knowledge management tool. jdom2 is a Java-based solution for accessing, manipulating, and outputting XML data from Java code. lang3 is a several utility classes with null-safe methods for String

[4]http://www.kclee.de/clemens/java/javancss/

parsing and manipulation. NanoXML is a small non-validating parser for Java. Finally, Joda-Time provides date and time library for Java.

These projects include classes which are complex and represent cases where SBST approaches, such as EvoSuite, have experienced problems in achieving high branch coverage [34][36]. For example, the constructor of PathURN class, which is in the Conzilla test subject, calls the URN superclass, and requires a valid URN string to produce the desired PathURN object. As a result, without a desirable PathURN object, a program execution throws an exception before any of the code in PathURN can actually be executed.

Such classes are ideal for use in evaluating GAMDR, since they represent inheritance and implementation relationships. We expect that GAMDR with its ability to precisely extract necessary dependency relations among the classes and identify the root cause leading to the creation of a PathURN object, can potentially guide the search to produce desirable PathURN objects and achieve high code coverage.

4.4.2 Experimental Setup

This section describes our evaluation setup in order to evaluate the effectiveness of GAMDR, and describes how GAMDR, simple GA (i.e., without MDR enabled), RT, and EvoSuite are configured.

We used identical configurations for simple GA and GAMDR to ensure as fair a comparison as possible between GAMDR and simple GA. They both use the same fitness function defined in equation (1), and use a single point crossover with probability 0.8. Mutation probability of an individual is 0.9, and the probability of insertion, changing, and removing statement is set to $r = 0.01$. 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. The choice of these parameters that we used in our evaluation were commonly adopted and used in several testing search-based approaches [13][16][22][30].

We also used EvoSuite version 20130910 with the default configuration because prior works showed that tunning EvoSuite with different parameters failed to outperform the default configuration [10][65].

To compare random testing (RT) with the search-based approaches, we adopted the proposed approach by Ciupa *et al.*[23], and implemented random generation with the following configurations: the length of test cases in RT is set to 200 [40][41], 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 (0x200x7E) [74].

As our primary purpose of this study is to evaluate the capability of each testing approach in achieving high branch coverage, the main challenge is identifying a fair stopping criteria to all testing approaches. Search algorithms are commonly compared in terms of the number of fitness evaluations per target branch, and this metric was used in different previous studies such as [50][74]. In this particular evaluation, because all the testing approaches target all branches at the same time, we decided to set up a fixed time budget for each testing approach including the early static analysis and instrumentations stages. We also applied each approach with time limit of 5 minutes to show effectiveness of GAMDR in achieving high branch coverage.

Our secondary goal of this study is to compare the efficiency of GAMDR with the other testing approaches. To achieve this, we specifically chose 30 seconds as a time

limit for each class under test (CUT) [41]. A budget of only 30 seconds provides a quick insight into how the search process finds useful data input and quickly achieves a high branch coverage [41].

To evaluate the statistical difference of GAMDR, we followed the guidelines in [9]. We implemented JaCoCo Version 0.7.5[5] to measure coverage during test generation. In addition, to reduce the randomness of each testing approach, it is important to conduct the experiments multiple times. Therefore, for each CUT (not per test subject program), we ran each testing approach 30 times for each time limit (i.e., 30 seconds and 5 minutes) with different random seeds.

Finally, all the experiments in the evaluations were conducted on a machine with Intel(R) Xeon(R) CPU E31240 V2 @ 3.40GHz and 14 GB RAM, running Red Hat with Kernel Linux 2.6.32.

4.4.3 Effectiveness of GAMDR

RQ 1: How effective is GAMDR in achieving branch coverage in comparison with representative test generation approaches? Table 4.2 summarizes the branch coverage percentage which is averaged out of the 30 runs of each testing approach at a time limit of 5 minutes. In the table, the highlighted values with bold text indicates that a particular testing approach obtained the highest coverage (IF STATISTICALLY SIGNIFICANT) for that test subject. The p-values are based on the Mann-Whitney-Wilcoxon test at a level of α=0.05, and performed with R version 3.0.1[6].

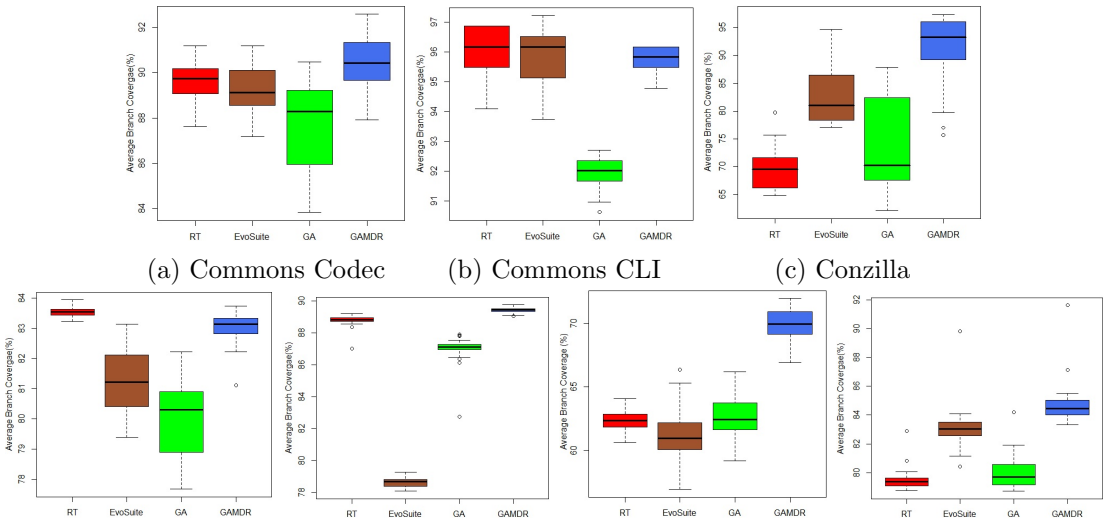
In addition, to visually compare all testing approaches, Figure 4.4 shows the box-plots

[5]http://eclemma.org/jacoco/
[6]https://www.r-project.org/

Test Subject	RT(%)	EvoSuite(%)	GA(%)	GAMDR(%)
Commons Codec	89.71	89.28	87.76	**90.47**
Commons CLI	95.96	95.67	91.97	95.81
Conzilla	70.05	82.79	73.78	**91.85**
Jdom2	**83.58**	81.22	80.02	83.03
lang3	88.48	78.64	86.98	**89.43**
NanoXML	62.87	61.34	62.51	**69.88**
Joda-Time	79.52	83.19	79.95	**85.10**

Table 4.2: Branch Coverage achieved by RT, EvoSuite, GA, and GAMDR at 5 minutes

and compares the actual obtained branch coverage of RT, EvoSuite, GA, and GAMDR at 5 minutes for each test subject.



(a) Commons Codec (b) Commons CLI (c) Conzilla

(d) Jdom2 (e) lang3 (f) NanoXML (g) Joda-Time

Figure 4.4: Coverage results over 30 runs of each approach on each of the 5 test subjects ar 5 minutes

4.4.3.1 Comparison with Random Testing (RT)

It is clear from Table 4.2 and Figure 4.4 that GAMDR outperforms RT on Conzilla and NanoXML subjects in branch coverage, but GAMDR achieves little coverage increase over RT in two subjects, namely Commons Codec and lang3. For the final test subject program, Jdom2, GAMDR almost achieves the same branch coverage as RT.

As shown in Table 4.2 and Figure 4.4, GAMDR achieves 91.85% branch coverage for the subject Conzilla and 69.88% branch coverage for the subject NanoXML, while RT achieves 70.05% and 62.87% branch coverage, respectively. One major reason GAMDR achieves higher branch coverage than RT is that most Conzilla and NanoXML test subjects contain classes containing constructors that call superclasses and require calling methods that are in a correct order that have valid arguments. Due to these requirements, most randomly-generated sequences of method calls may fail to generate a valid object instance of the class under test and then fail to reach desirable states and cover target branches.

For example, in the Conzilla test subject, the constructor of the class `identity.URL` calls the `identity.URN` superclass. Each of these classes has a constructor that contains nested branches that require specific character values, such as ':', '/', and '/'. These constraints make the task of generating a valid object instance of the `identity.URL` class more difficult. Although RT randomly generates a new sequence of method calls on each iteration with a length of 200, which means that RT has a higher probability of calling the constructor of `identity.URL` class. However, RT frequently fails to find the correct character values that satisfy the constraints and cover the branches in the constructors of the classes. As a result, RT can not create a valid object instance of the `identity.URL` class. In fact, most of the test data generated by RT could not

reach beyond 8 branches out of 22 and only achieved 37.12% branch coverage of the `identity.URL` class.

In contrast, the static analysis used in GAMDR helps identify the root cause which leads to failure to generate instances of the `identity.URL` class, and identifies all relevant accessible constructors of the `identity.URL` class that can be used to generate valid object instances. MDR enhances the search process by allowing GAMDR to concentrate on not only creating new instances of the `identity.URL` class more frequently, but also focusing more effectively on the valid existing instances of `identity.URL` class. Because of these advantages, GAMDR achieves 87.42% branch coverage of the `identity.URL` class, which is 50.30% more than RT.

The results also show the test cases generated with GAMDR achieves little coverage increase over RT in two test subjects, namely Commons Codec and lang3. However, a detailed analysis at the class level reveals that GAMDR has higher coverage than RT on large classes. For example, in lang3, the class `lang3.BooleanUtils` contains 40 public methods and 250 branches, 101 of which are nested in one public method namely `Boolean toBooleanObject(final String str)`. GAMDR achieves 89.4% branch coverage of the `lang3.BooleanUtils` class, whereas RT only achieves 84.8%. The primary reason is that the significant number of public method and classes decreases the probability of picking relevant methods in a test sequence calls. Consequently, RT encounters difficulties generating required sequences of method calls to cover nested branches. However, the proposed static analysis helps GAMDR to reach and cover more branches by testing only related methods together.

For the final test subject, Jdom2, RT actually achieves slightly increase in the branch coverage with an average equal to 0.54%. The reason is that most methods in Jdom2 do not have any constraints or dependencies between each other. Therefore, randomly

generating of sequences of method calls can achieve high branch coverage.

In general, our results show that GAMDR is more effective than RT on large size classes, suggesting its strength lies in testing classes requiring generate method calls in specific orders with specific arguments, a weakness of random testing observed in previous studies [40][92].

4.4.3.2 Comparison with EvoSuite

Results in Figure 4.4 and Table 4.2 answer **RQ1** by clearly showing, with high statical confidence, that GAMDR outperforms EvoSuite on 3 test subjects, namely, Conzilla, lang3 and NanoXML. There are two major reasons for achieving these results. First, the optimizations used by EvoSuite, such as constant seeding (i.e., constants are collected statically and dynamically at run time) lacks necessary directions towards the appropriate branches from which they are collected. As a result, EvoSuite is unlikely to pick up specific constant values from the potential seeding pool to execute certain branches. Second, the randomize of the mutation operator in EvoSuite lacks necessary guidance to those methods and parameters where changes may be most required to execute certain branches.

Figure 4.4 and Table 4.2 also show that GAMDR achieves little coverage increase over EvoSuite in Commons Codec and Jdom2. These test subjects do not require a complex of sequence of method calls in order to cover certain branches. In order to understand the effectiveness in terms of branch coverage between GAMDR and EvoSuite, we select and manually analyze the code coverage of four classes where a significant difference is observed. Figure 4.5 shows the percentage of branch coverage achieved by both GAMDR and EvoSuite on the selected classes.

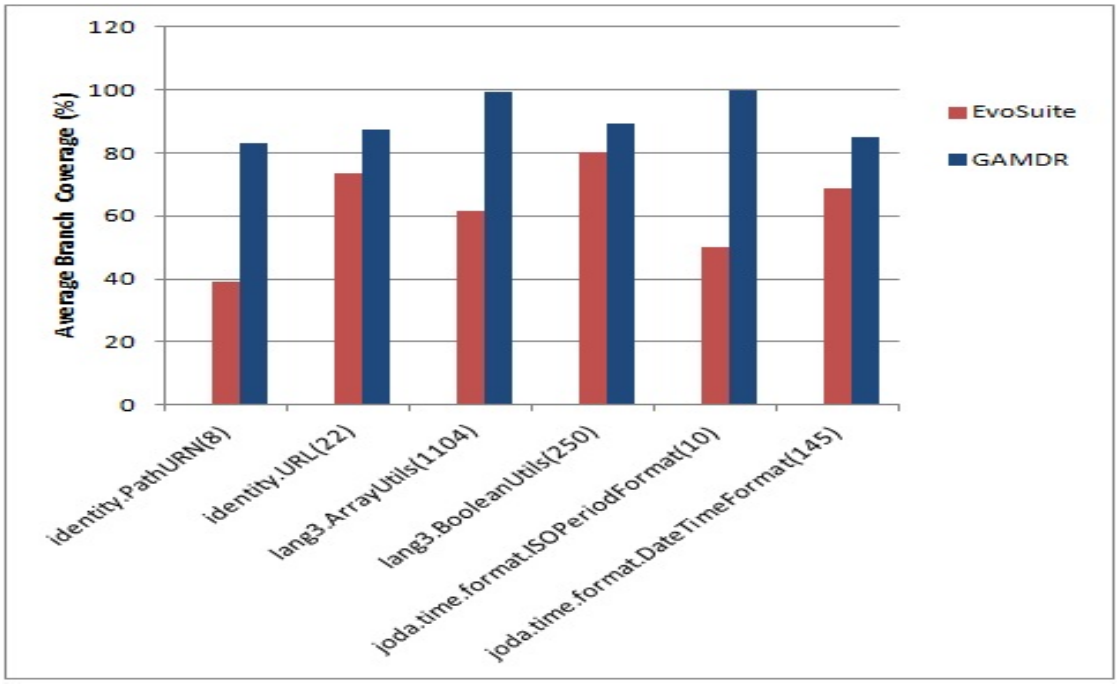


Figure 4.5: Comparison of EvoSuite and GAMDR on classes in terms of branch coverage

As shown in Figure 4.5, GAMDR achieves higher branch coverage in small classes, such as `identity.PathURN` and `identity.URL`. For example, in the Conzilla test subject, GAMDR achieves 82.92% branch coverage for the `identity.PathURN` class, which contains only 8 branches, while EvoSuite only achieves 39.17%. The reason for this low coverage is that the constructor of the class `PathURN` extends class `URN` which also extends class `URI`. These inheritance relations increase the size of the search space since the test cluster includes all the set of available classes, their public constructors, methods, and their parameters. We speculate that EvoSuite generates test sequences that might have included irrelevant constructor or method calls in the initial population. Since EvoSuite relies on mutation operations to modify a test case with a certain probability, EvoSuite could no longer produce new offspring (i.e., new test cases) with better

performances than their parents due to the randomized operations in the mutation operator. EvoSuite could not preserve the population diversity during the evolution, and as a result, the search was trapped in one area of the search space (i.e., local optimal [68]).

Figure 4.6 shows another example where the static analysis used in GAMDR helps to identify methods dependence relation based on the fields they write i.e., *write-read relation*, and prefers *write-read* related methods during the mutation operations to cover a target branch.

```
1.   public class ISOPeriodFormat {
2.     private static PeriodFormatter cStandard;
3.    ...4 more private member fields ...
4.     public static PeriodFormatter standard() {
5.         if (cStandard == null) {
6.              cStandard = new PeriodFormatterBuilder();
7.              ...more statements ...
8.         }
9.     return cStandard;
10.    }
3.    ...4 more public static methods...
11.  }
```

Figure 4.6: ISOPeriodFormat class

As is shown in Figure 4.6, the branch condition at line 6 uses a *null* value and so the branch distance measurement does not provide any guidance, i.e., flag problem [68]. As a result, EvoSuite blindly tries to mutate the test cases and faces challenges in generating sequences that require to call the same method multiple times to produce the desired object state for the field member **cStandard**. EvoSuite easily covers the branch that requires the *null* value and faces challenges in generating a sequence of method calls that call the **PeriodFormatter** method at least twice. In contrast,

the static analysis uses in GAMDR allows to identify methods that change the member field **cStandard**. Thus, during the mutation stage, GAMDR recommends to invoke the same method before **PeriodFormatter** method, since they both have *write-read relation*, and testing them together allows to change the state of the member field **cStandard**. This permits GAMDR to generate more effective sequences of method calls that allow to cover all the target branches and reaches 100% branch coverage of the class `joda.time.format.ISOPeriodFormat`.

The Figure 4.5 also shows the improvement in branch coverage between GAMDR and EvoSuite in large classes, such as `lang3.ArrayUtils` and `lang3.BooleanUtils`. GAMDR achieves 99.29% for the `lang3.ArrayUtils` and 94.26% for the `lang3.BooleanUtils`, while EvoSuite only achieves 79.98% and 89.4%, respectively.

Manual analysis for both classes shows that each class contain a large number of public methods, each of which contains a large number of branches. For example, the `lang3.ArrayUtils` class includes 233 methods and 1104 branches, and the `lang3.BooleanUtils` class includes 40 methods and 250 branches. Most of the branches require `null` values or character values. Intuitively, we would expect that EvoSuite should achieve higher coverage, as the seeding strategy would influence the search and help to cover not-trivial branches [28]. However, due to the large size of the search space EvoSuite can not identified of which method calls or parameters need to be mutated to cover certain branches. The results indicate that MDR is indeed useful in helping to increase branch coverage by identifying relevant methods and parameters that need to be mutated in order to cover particular target branches.

4.4.3.3 Comparison with simple GA

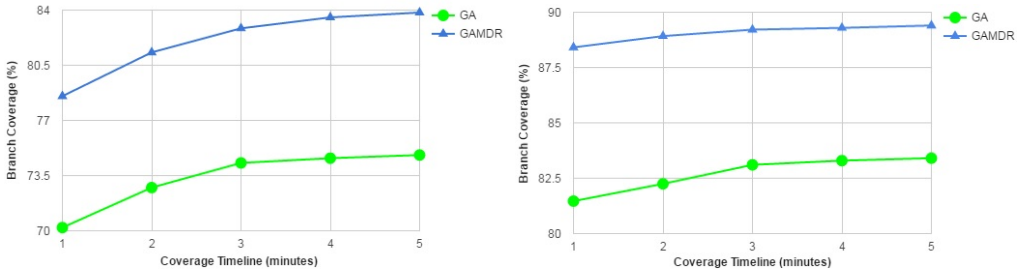
Our results show that GAMDR achieves higher branch coverage than simple GA for all five test subjects. This result is easily explained by considering the way GA works. GA uses a genetic algorithm which maintains a population of individuals (i.e, test cases), and randomly changes , with a certain probability, particular parts of the test cases (i.e, mutation operations). These random mutation operations can lead to losing the diversity of the population, and the search thus can be easily trapped in local optima [71].

To prevent the search from being trapped in local optima, GAMDR utilizes MDR to discard irrelevant methods and constructors calls, which do not improve branch coverage, and directs the mutation operator to those parts of the test cases where changes are most beneficial. These directed changes are most likely to improve the exploration of the search space, maintaining diversity among the individuals of the population, and hence improving the branch coverage.

We next investigated the impact of MDR on the performance of GAMDR by analyzing the code coverage at the class level. We chose two classes from Commons Codec and lang3 test subjects, the `language.DoubleMethaphone` class, and `lang3.BooleanUtils` class, respectively. Both of these classes contain a large number of conditional statements and public methods. Class `language.DoubleMethaphone` contains 443 branches and 38 public methods, whereas `lang3.BooleanUtils` contain 250 branches and 40 public methods. In addition, the difficulty to cover a large number of the branches in these classes lies in the generation of constant values, such as null values. These type of branches cause losing of useful branch distance information, and thus do not provide good guidance to the fitness function that the search can use to find test inputs [91].

Figure 4.7 compares the average branch coverage for `language.DoubleMethaphone`

and `lang3.BooleanUtils` at one minute intervals over five minutes.



(a) language.DoubleMetaphone (b) lang3.BooleanUtils

Figure 4.7: Branch coverage comparison between GA and GAMDR over 5 minutes with one minute intervals

As shown in Figure 4.7, the use of MDR is beneficial for the search process, GAMDR achieves 83.84% branch coverage for `language.DoubleMethaphone` and achieves 89.4% branch coverage for `lang3.BooleanUtils`, while GA only achieves 74.80% and 83.4%, respectively. The lower coverage of GA can be attributed to the poor search guidance that the fitness function provides. With a poor fitness function, the search are not capable of finding and producing better individuals in large search space, such as the space of possible inputs for `language.DoubleMethaphone` and `lang3.BooleanUtils` classes. To understand this, we should consider the problem of premature convergence in genetic algorithms [66]. This phenomenon occurs when a few better (i.e, fitter) individuals quickly dominate the population, thus reducing the population diversity and causing the search process to be trapped in a local optima [68][66]. Applying the mutation operator can preserve the population diversity and prevent the search process from being trapped in local optima [68].

In contrast, the use of MDR improves the performance of the mutation operator by identifying irrelevant constructor, method, and parameter calls, and excluding them from the search space, so the search process focuses only on method and parameters calls where

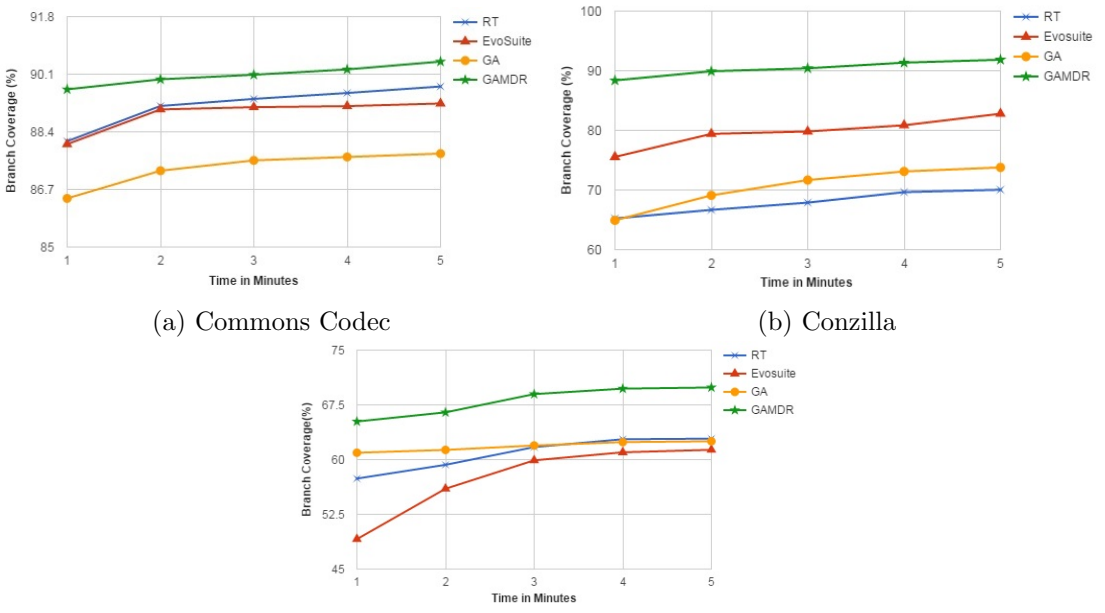
changes are more likely to produce a fitter offspring. Hence, GAMDR retains population diversity and easily generates test sequences that achieve higher branch coverage than GA.

In summary, the results of figure 4.4 and table 4.2 show that in all test subjects there is a significant branch coverage improvement over simple GA. Combining MDR with GA helped GAMDR to generate test data that achieved a high level of branch coverage.

4.4.3.4 Improvements over Time

Five minutes can be a long time for a test data generation approach, and it may be the case that the improvement of GAMDR only appears after spending a large amount of time [36]. We studied the performance of the GAMDR and the other test data generation (i.e., RT, EvoSuite and GA) approaches at different time intervals. To this end, we performed a set of experiments and kept track of achieved branch coverage at each minute interval. The experiments were conducted on Commons Codes, Conzilla, and NanoXML test subjects. We chose these test subjects because these projects contain different types of classes. For example, Commons Codes has very few constraints on its class constructors and methods. The Conzilla and NanoXML contain more complex classes and represent cases where RT, EvoSuite and GA have problems in achieving high branch coverage. Each experiment was performed with the same configurations chosen in **RQ1**.

Figure 4.8 shows the time analysis for each test subject individually. The results show that, regardless of the time, there is always a large gap between the GAMDR and the other test approaches. Figure 4.8 also clearly shows that the beneficial of MDR does not appear overtime, but applies from the beginning.



(a) Commons Codec

(b) Conzilla

(c) NanoXML

Figure 4.8: For the three test subjects, average coverage at different points in time for the GAMDR, RT, GA, and EvoSuite at 5 minutes.

Figure 4.8 provides further insight and shows different behaviors among test subjects. For example, in Commons Codec test subject, RT performs very similar to EvoSuite until minute two where the performance of RT increases through time. The reason is that class constructors in this project do not need specific arguments. As a result, RT generates many legal sequences of method calls that create new object instances more often, while EvoSuite tends to focus on the existing ones. As a result, RT covers more branches over time.

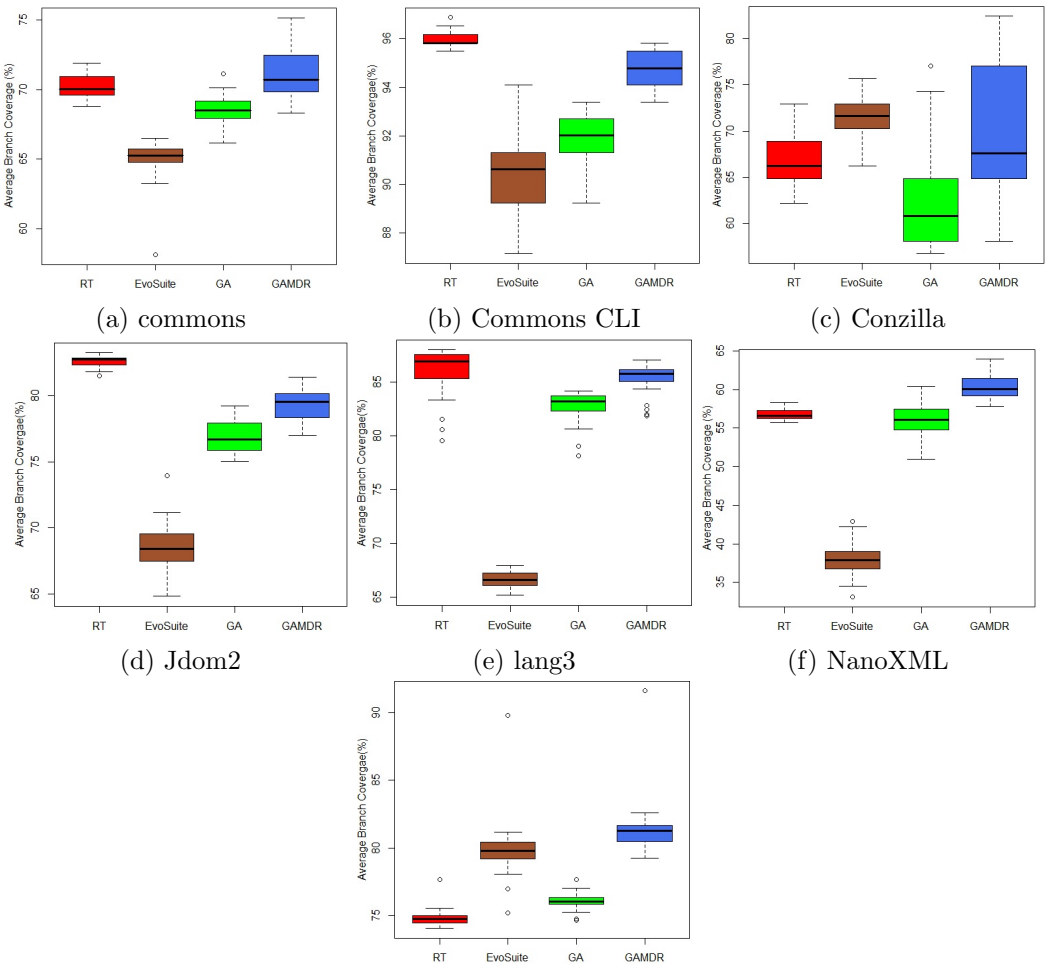
4.4.4 Efficiency of GAMDR

RQ 2: Given a fixed search time, can MDR improve the efficiency of a simple GA in achieving branch coverage in comparison with representative test generation approaches?

Our secondary purpose was to evaluate the capability of MDR to improve the efficiency of a simple GA in achieving high branch coverage. An ideal option to compare the efficiency of each approach was to set up a fixed number of fitness evaluations for all test generation approaches [48]. However, such a comparison would favor EvoSuite. This is because EvoSuite represents a set of test cases (i.e., a test suite) as an individual [31], whereas in RT, GA and GAMDR, a test case represents an individual, such that the comparison of fitness evaluations would not be fair.

A second option was to set up a short fixed time for all test generation approaches to examine the efficiency of GAMDR, in terms of how quickly additional branches are covered. In this particular evaluation, we chose the second option. We determined a timeout value of 30 seconds because it was commonly adopted in testing approaches [41]. We also empirically observed that for some classes 30 seconds, would be more than enough to cover all the feasible branches.

Results of the experiment are presented in Figure 4.9. All results are averaged over thirty runs of each test subject. Figure 4.9 shows that GAMDR is more efficient than the genetic approaches, EvoSuite and GA, because it achieves the highest branch coverage for all test subjects, except for Conzilla. For Conzilla test subject, EvoSuite achieves the highest branch coverage among all test approaches at 30 seconds. The reason is that most of the classes in Conzilla perform string manipulation operations and most branches depend on characters (e.g., if(nuri.indexof(':')==-1)). Thus, test inputs generated by



(a) commons (b) Commons CLI (c) Conzilla

(d) Jdom2 (e) lang3 (f) NanoXML

(g) Joda-Time

Figure 4.9: Coverage results over 30 runs of each of approach on each of the 5 test subjects at 30 seconds

EvoSuite are more efficient than GAMDR to cover trivial branches due to the use of optimizations, such as constant and dynamic seeding [91].

Figure 4.9 also compares GAMDR and RT in terms of average branch coverage achieved at 30 seconds. GAMDR outperforms RT in the case of complex programs. Considering the NanoXML test subject, for example, GAMDR is more efficient than

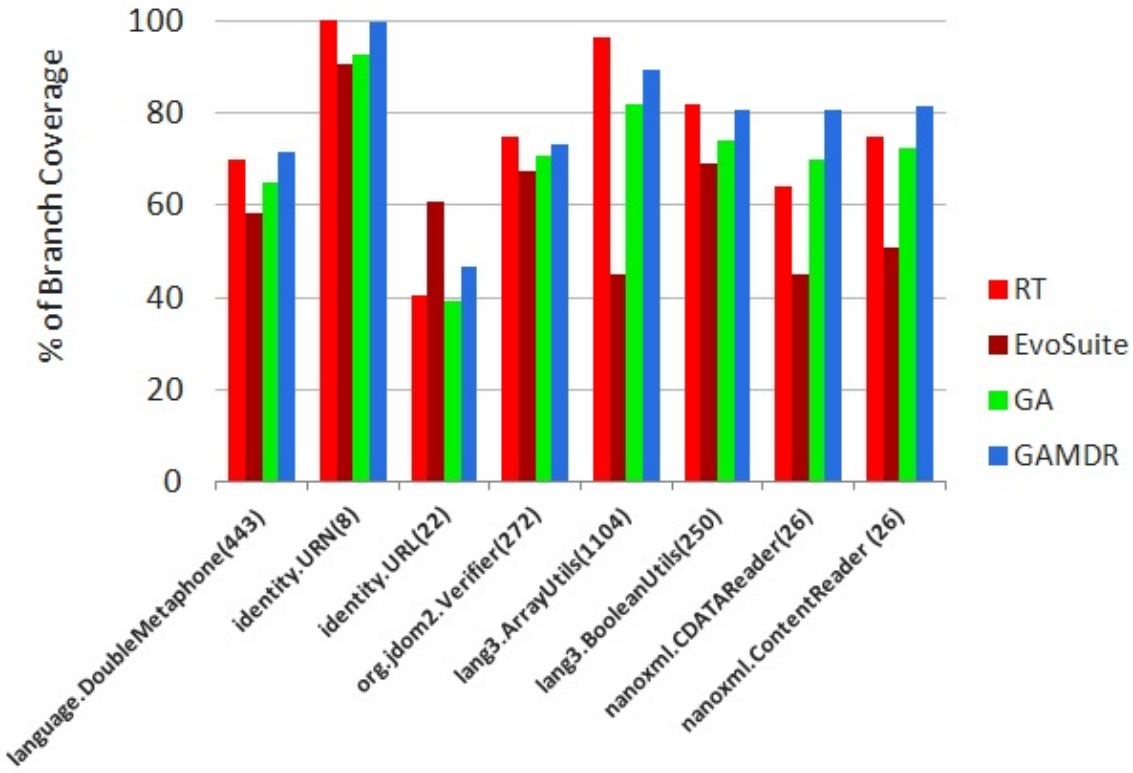


Figure 4.10: Comparison of RT, EvoSuite, GA, and GAMDR on selected classes in terms of branch coverage

RT because at 30 seconds it covers 60.23%, whereas RT covers 56.81%. For other test subjects, such as Commons Codec, Jdom2 and lang3, RT is more efficient than GAMDR and achieves higher branch coverage at 30 seconds. The primary reason is that RT repeatedly generates a sequence of method calls (i.e, test cases) with a length of 200. Thus RT has a higher probability to explore the search space wider and covers as many trivial branches as possible in a shorter duration of time. In contrast, GAMDR relies on the genetic algorithm and mutates an existing test case with a certain probability. As a result, GAMDR works longer to explore the search space and may, in time, fully cover all trivial branches.

To understand the observed difference of the branch coverage within 30 seconds, we selected different classes from these test case subjects. Figure 4.10 presents the percentage of the branch coverage achieved by each approach on the selected classes.

The results in Figure 4.10 provide evidence that GAMDR performs better than EvoSuite and GA in terms of efficiency. Specifically, GAMDR is more efficient than EvoSuite and GA on classes that contain a large number of branches. For example, GAMDR achieves 71.5% and 94.2% for the `language.DoubleMetaphone` and `lang3.ArrayUtils` at 30 seconds, whereas EvoSuite reaches 58.1% and 45.1 % and GA reaches 64.7% and 81.8 %, respectively. The primary reason is that GA, and EvoSuite approaches randomly choose which method to mutate (i.e, insert, update, and delete) from all methods available in a class under test. However, there is only a subset of methods that to be mutated in order to cover particular branches. In contrast, MDR allows GAMDR to efficiently increase the speed with which the search space is explored by guiding the mutation operator towards only relevant methods and covering and reaching more target branches in less time.

Note that for `identify.URL`, EvoSuite achieves higher branch coverage than GAMDR, GA, and RT. In this class, branches rely on character constants. Therefor, the seeding used in EvoSuite helps to cover branches more quickly than GAMDR, GA, and RT.

In **RQ 2**, the question addresses whether MDR can help GAMDR to avoid the exploration of useless sequences of method calls and generate optimal sequences for the class under test (i.e., test cases that cover all the target branches). Thus, GAMDR converges faster to the optimal tests and reaches the global optima for the class under test faster than the other test approaches.

To this end, we further investigated the `identify.URN` class in the Conzilla test subject, since GAMDR and RT covered all the target branches within 30 seconds. Figure

4.11 impressively demonstrates how MDR helps GAMDR to converge quickly to the optimal solutions and find the global optimum for the class `identify.URN`. GAMDR covered all 8 branches in average 4.09 seconds, while RT took a 7.26 seconds to cover all the target branches for the class `identify.URN`. Each test data inputs generated by both GA and EvoSuite missed one branch compared to test data generated by GAMDR. As a result, and contrary to the random of mutation operator, our approach GAMDR achieves a high code coverage in less than 30 seconds.

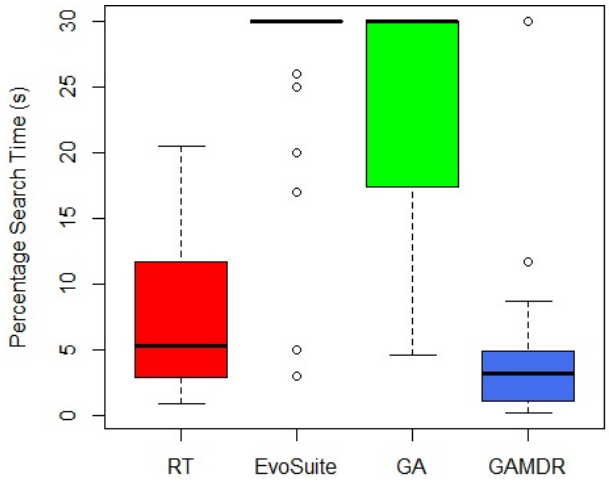


Figure 4.11: Comparison of RT, EvoSuite, GA, and GAMDR on identify.URN class in terms of branch coverage

To summarize, our results show that the static analysis used in GAMDR leverages the information of how relevant methods are used and identifies data dependences affecting the execution of branches. This information is used to help GAMDR to effectively and efficiently improve the exploration of complex and large search space by recommending related methods to test together for quicker high branch coverage achievements.

4.4.5 Difficult Branches

RQ3. What types of branches did GAMDR fail to cover? Although GAMDR achieved higher coverage than RT, EvoSuite and GA, the coverage achieved is still not 100%. We identified some branches that are difficult to cover, as the following:

- **String and special characters comparison**. Some branches require well-formated string or special characters value to be covered. If a test generator fails to generate these values, the generated test will fail to cover these branches. For instance, the class Verifier from Jdom2 test subject contains a significant number of branches that depends on special characters, such as '\t ', '\n ', or '0x00B7 '. It is difficult for GAMDR to randomly generate a required character to cover such branches. In future work, we plan to address this issue by seeding the constants during the search process that can help GAMDR to cover branches which require particular string values or special characters values.

- **Environment dependency**. Another reason for the low coverage achieved by GAMDR is that some classes contain constructors or/and methods, which takes environment variables or specific file structures. For example, in the NanoXML test subject, classes, such as XMLWriter or StdXMLReader, contain constructors and methods that require a parameter that must be referring to an existing XML file. GAMDR can not automatically generate these types of inputs. However, there is a lot of research on how to handle environment dependencies [11][81], which is currently beyond the scope of our research.

4.4.6 Threats to Validity

As with any empirical study, there are several potential threats impact to the validity of our conclusions.

- **Construct validity threats** refer to the degree of the relationship between theory and the way we judge the performance of our technique. In our study, we compared GAMDR with other testing approaches, such as RT, GA, and EvoSuite, in terms of branch coverage. We gave priority to the branch coverage because it is the most common criteria to measure the performance of a testing technique [51]. Another measure of performance, one would compare search algorithms based on the number of fitness evaluations [28]. A test suite in EvoSuite represents an individual, whereas GAMDR, GA,and RT, the test case, which has different lengths, represents the individual. Thus, the comparison of fitness evaluations would favor EvoSuite. As a result, we chose time limit (i.e., 5 minutes and 30 seconds) rather than the fitness of the evaluations.

- **Internal validity threats** concern how the empirical process was carried out. One potential threat is that comes from an existing fault in our instrumentation and testing frame. To reduce these threats, we carefully tested GAMDR, and manually inspected the results of each component of GAMDR. Furthermore, the randomized nature of any testing algorithm may affect the internal validity. As a result, we ran each testing approach multiple times (i.e., 30 times) with different seeds. We also followed rigorous statistical procedures to analyze the results [9].

- **External validity threats** concern of the generality of our conclusions. We used a large number of test subjects, which contains 175 classes. All the test subjects and

classes have been used in previous empirical studies [28][31][74][85]. Another threat might come from not trying different parameter settings for each test approach. To reduce this threat, we carried out our experiments with the most common parameter settings which have been used in previous studies [10][13][16][41].

4.5 Conclusion

This chapter has introduced and evaluated GAMDR approach for searchbased software testing (SBST). GAMDR applies a genetic algorithm, augmented with method dependency relations for improving branch coverage for Java programs. GAMDR leverages MDR to reduce the size of the search space and direct genetic mutations on particular parts of the test cases for certain branches to be covered.

Our empirical study shows that GAMDR effectively and efficiently finds test data inputs and achieves high code coverage in less than 30 seconds. The empirical study also shows that GAMDR achieves higher branch code coverage than random testing (RT), EvoSuite, and a simple GA (without MDR enabled) for complex Java programs.

As mentioned in the Section 4.4.5, further improvements are required in order to improve branch coverage. This is the subject of the next chapter.

Chapter 5: A hybrid Search (A Memetic Algorithm)

5.1 Introduction

Memetic algorithms (MAs) [77] are a metaheuristic that combines (hybrid) both global and local search (e.g., a genetic algorithm (GA) with a hill climbing (HC)). A simple way to implement a MA is to employ a stage of local search to improve each individual at the end of each generation [53].

This chapter presents an automated search-based technique that uses a memetic approach. The memetic algorithm combines both a genetic algorithm (GA), described in Chapter 4, 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 class under test (CUT), while minimizing the length of each test case. The latter is used to target uncovered branches in the preceding search phase. However, two important changes are made to optimize individuals (i;e, test cases) that allow the search to execute more target branches. Firstly, the HC terminates for each individual upon reaching a local optima [53]. Secondly, the HC is employed the MDR to diversify the search and explore new and unseen areas of the search space [53].

The main contributions of this chapter 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). We also employ MDR with HC to improve its effectiveness.

2. We also introduce a way to seed constants into the search process when targeting uncovered branches.

3. 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 chapter is organized as follows.

5.2 Applying Memetic Algorithm (MAMDR)

In this section, the concepts of our hybrid search-based approach are presented. Figure 5.1 illustrates our approach, called MAMDR[1], architecture.

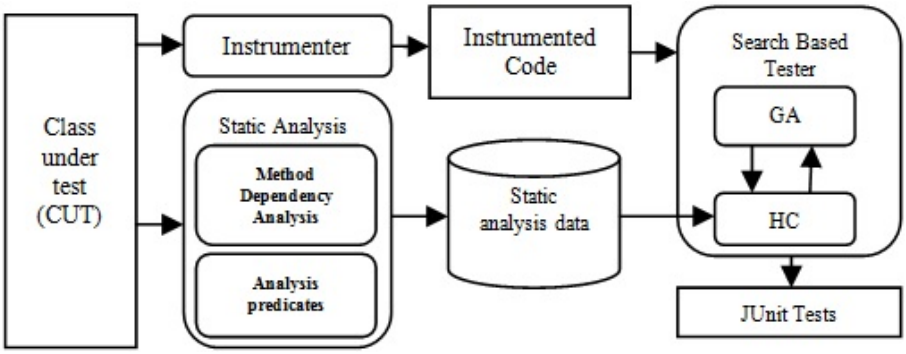


Figure 5.1: Overview of MAMDR

As shown in Figure 5.1, MAMDR is built on top of the GAMDR framework for search-based testing. In contrast to GAMDR, it extends the Genetic Algorithm (GA) behind the GAMDR test generation tool into Memetic Algorithm, by equipping it with

[1]The name is derived from improving Memetic Algorithm via Method Dependence Relations

a local search (HC). In addition, the HC is extended by including MDR to generate sequences of related method calls and also initializes values using constants collected from the source code that would cover the target branches. Let us detail how these extensions all fit together.

5.2.1 Genetic Algorithm

The GA used in MAMDR described in Chapter 4. However, we use the fitness function in equation (1) to guide the GA search, and it is combined two objectives in a single function [13].

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

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 [13].

5.2.2 Hill Climbing (HC)

When the GA results stagnate, we employ hill climbing (HC) as the local search, similar to that used in the works [13] and [16]. 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.

Fitness Function: The fitness function that guides HC is similar to that which was used by Arcuri and Yao [13]. 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.

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

Where k is a normalizing function, and we use the normalization function [8]: $k = \frac{x}{((x+1))}$, 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 $x = 10 - 2$ [31]. Finally, we integrate BD with the total branches coverage of the individual to guide the HC search in the following way:

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

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 [13]. 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 branchs 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 conditions target branch:

(a) **Member field:** To deal with class member fields, we followed a similar approach to the ones used by Thummalapenta et al. [94]. 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 [48]. In spite of that, Harman et al. [48] 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. [4]. 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 [16]. (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.

5.3 Evaluation

To validate our approach described in this chapter, we compared its effectiveness against three different approaches: pure random testing, the EvoSuite [31] tool as a representative for search-based approaches, and a simple MA [13] [16]. 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. [23]. In addition, we compared a simple MA without method dependency rela-

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

Table 5.1: Details of the test subjects used in the empirical study.

tion (MDR) with our approach to show the effectiveness of our search space reduction approach in test data generation. MA uses both GA and HC. 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 [16] and Seeker [94]. 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 [16]. 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 [30]. We also included DateTimeFor-

mat 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. [92], to see whether our approach has a negative effect on the performance of the search process when its power is not needed. Table 5.1 lists our evaluation subjects, including their number of public classes, lines of code , and number of instrumented branches.

5.3.1 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).

5.3.2 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 = 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 [41]. 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 (0x200x7E) [74]. All the experiments were conducted on a machine with Intel Core 2 Quad CPU @ 2.66 GHz and 8 GB RAM.

To evaluate the statistical difference of our approach, we followed the guidelines in [9]. 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).

5.4 Results

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

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

Table 5.2: Average branch coverage Achieved by RT, EvoSuite, MA, MNS, and MWS.

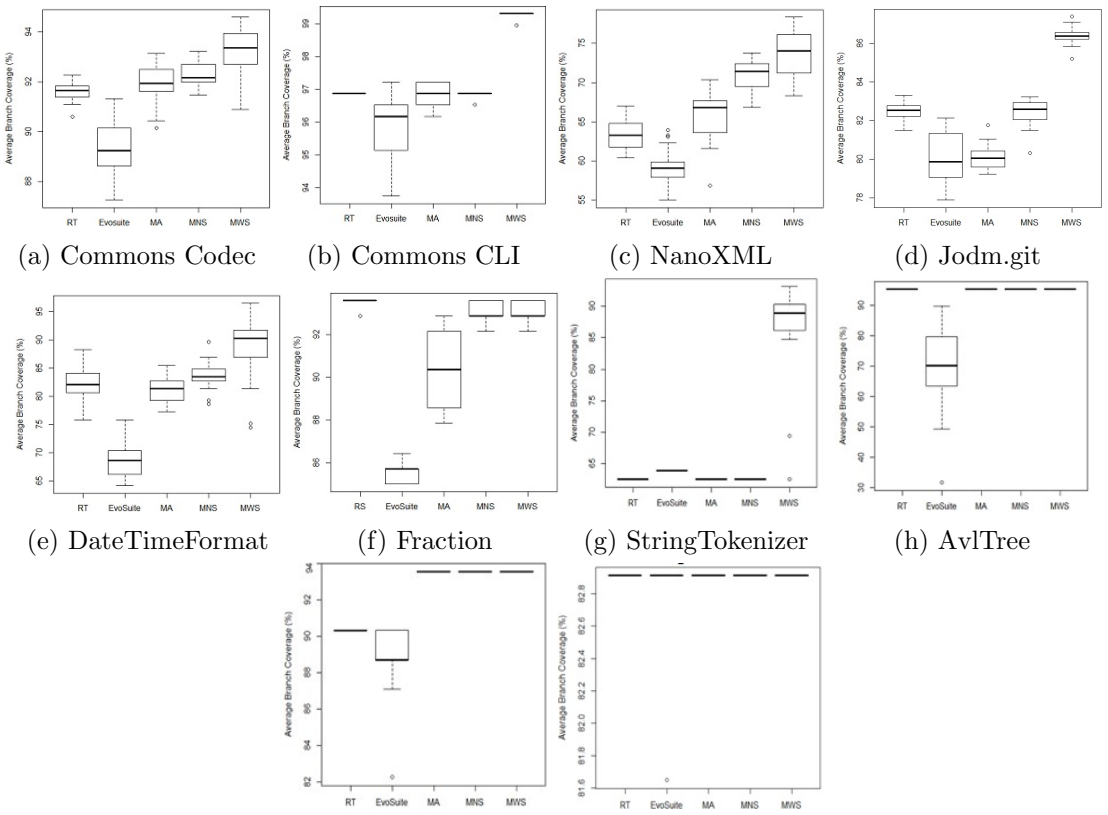
5.4.1 Coverage Results

Table 5.2 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.

Figure 5.2 shows a box-plot of the actual average branch coverage achieved over 30 runs of each approach on each test subject.

5.4.2 Comparison with RT

The results in Table 5.2 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 ex-



(a) Commons Codec (b) Commons CLI (c) NanoXML (d) Jodm.git

(e) DateTimeFormat (f) Fraction (g) StringTokenizer (h) AvlTree

(i) BinomialHeap (j) TreeMap

Figure 5.2: Average Branch Coverage of each of the 5 approaches on each test subject

plained 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 [105]. 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 [16]. In our experiments, the length of RT test cases was set to 200.

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 [16].

5.4.3 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 [31], 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 [34]. 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 [34]. 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 [34] [36] [81].

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 [48].

5.4.4 Comparison with MA

Table 5.2 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 [105]. 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.

5.4.5 Impact of seeding constants

As might be expected, seeding improves branch coverage 7 of 10 test subjects (Table 5.2). 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.

As the Figure 5.2 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 [34]. 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 [40], which is three times as long as EvoSuites. Although it may be possible to find parameter settings for which EvoSuite performs better, discovering parameter settings can be considered computationally expensive [10].

5.5 Threats to Validity

As with any empirical study such as this, there are several potential threats impact to the validity of our conclusions. fThis section provide a brief overview of the threats to validity and how they have been addressed.

- **Internal validity threats** 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.

- **External validity threats** 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.

5.6 Conclusion

In this chapter, 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 Object-Oriented 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 MAs.

Chapter 6: Strengthening the Genetic Algorithm with Initial Population and Seeding

6.1 Introduction

In recent years the number of papers on search-based techniques has been increasing quite rapidly, and the majority of papers focus on the use of search-based techniques, such as the genetic algorithm (GA), in automated generation of test cases for structural coverages [55], as in the case of object-oriented programs (e.g., [1] [13],[16][31]). Although GA has been shown to be a promising approach for the task of software testing, many different factors affect the performance of the GA. These factors include the population size, genetic operators, initial population, and seeding.

The aim of this chapter is to discuss and evaluate the following strategies: a strategy to generate the initial population and a strategy to seed constants extracted from the source code. The goal is to improve search-based test generation for object-oriented programs and allow the search to generate tests more effectively. Another objective is to enhance the performance and ability of the GA to reach the global optimum instead of being stuck in a local optimum.

This chapter introduces two primary contributions:

1. A new strategy to set the initial population is applied to improve the quality of the initial population of the GA, and to reach the optimum solution in the search search space.

2. A constants seeding strategy is applied to increase the likelihood of covering a target branch.

3. An empirical study is performed to show the effectiveness of the proposed strategies on the performance of the GA. Although, we focus on the branch coverage criterion and Java language, our approach can be extended to other coverage criteria and used object-oriented languages.

The rest of this chapter is organized as follows. Section 6.2 introduces two strategies of improving the initial population, and Section 6.3 sets up the context of seeding constants strategy. Section 6.4 then presents an empirical study and discusses our findings. Section 6.5 discusses threats to validity of this study. Finally, Section 6.6 concludes the chapter.

6.2 New strategies for the initial population

A genetic algorithm (GA) usually starts with a set of possible solutions (i.e., test cases) randomly generated as an initial population. Although, the process of randomly selecting all solutions of the initial population from the search space of the problem seems simple, the initial population critically affects the convergence, the capability, and the performance of the GA [96]. If the size of the search space is small, these properties of the GA may not be affected. However, for most of object-oriented programs, the search space is large and complicated because of two reasons. First, generating a sequence of method calls often involves methods from multiple classes, resulting in a large space of candidate method calls [94] [96]. Second, finding a desired object of a class, and finding a proper sequence of method calls to put the classes in a desirable state, is difficult due to its large search space [94]. Thus, domain knowledge information can be exploited to

reduce the size search space and improve the quality of the individuals that are able to cover many target branches of the CUT.

Given some domain knowledge information, it can be easy to create an initial population that produces better starting points and to increase the diversity for the search, which may impact the increasing-coverage capability of the GA [28].

Several works have been proposed to improve the initial population of the GA [28] [76] [96]. Fraser and Arcuri [28] implemented a simple strategy, called "AllMethods" to maximize the number of method calls in the initial population. In their strategy, during the initialization, each time a new method is inserted into the population, the next method is chosen based on a ring buffer of all the methods. In other words, each time a new method call is inserted, it is not chosen randomly; instead, the next method in the buffer is returned. The aim of this strategy is to guarantee that all the methods of the class under test (CUT) are called in the initial population. Their results showed that the technique improves the branch coverage for different classes, particularly for classes containing a high number of methods.

Miraz et al. [76], also improve the initial population by performing random testing to generate individuals and selecting the best ones out of a larger pool to generate the initial population. They evaluated their technique on six Java classes and their experiments show that it increases the branch coverage for more complex classes, such as Red-Black Tree class [76].

In this chapter, we propose two new strategies to initialize the initial population of the GA. Unlike previous works, we adopt two different approaches in the design of the strategies: (1) The method dependency relations (MDR) given by Zhang et al. [105], and (2) The SWARM testing approach given by Groce et al. [44]. The intention is to improve the performance and effectiveness of the GA in terms of code coverage.

6.2.1 Directed-Population Strategy

In the case of generating test cases for object-oriented programs, there are three prerequisites to achieve full or at least high coverage (such as branch or statement coverage): (1) instantiate the class under test (CUT), (2) generate a sequence of method calls to produce desired object states for the arguments, and (3) execute a method that may reach the target branches [85] [94]. However, when the initial population is generated entirely randomly from the set of accessible methods it is quite challenging to generate such sequences that satisfy the preceding prerequisites.

To mitigate this problem, this study introduces a strategy (which we call "Directed-POP") to improve the initial population by attempting to select methods that generate desired object states of the CUT and reach the target branches. The strategy is based on the proposed approach method dependence relations (MDR) by Zhang et al. [105], and includes the following steps. First, a static analysis is implemented to identify a set of relevant methods or constructors based on the data members they may read or write. Second, during the initialization, when generating a sequence of method calls, i.e., an individual, a new method is randomly selected, and then the static dependence information is used to determine the dependent methods that are likely to change the state of the selected method. Finally, the selected method and its dependent methods are used to form a sequence of method calls.

The intuition is that instead of allowing the GA to start from an initial population that is randomly generated, the static dependence information is used to create individuals that contain relevant sequences of method calls that may change the state of an instance of the CUT and reach the target branches. Based on our experience on applying search-based techniques such as EvoSuite [31], we observed that the branches in the

CUT are not covered due to the inability to generate target object states of the CUT. The main reason is that there is often a low probability of generating proper sequences of method calls at random to construct desired objects of the CUT and reach the target branches [94].

6.2.2 SWARM-Population Strategy

Groce et al. [44] proposed and evaluated an inexpensive approach, called SWARM, to improve the diversity of test cases generated during random testing. In the random testing approach, all of the features of the CUT, i.e., public methods and constructions, are available during the construction of each test case. The SWARM approach, in contrast, is based on a construct configuration that randomly chooses which features to include in each test case. The idea behind SWARM is that omitting some features increases the effectiveness of testing due to interactions between features [42][44]. Experimental results show that SWARM testing increases coverage and can improve the fault detection dramatically [44].

In the present study, we implement this strategy, which we call "SWARM-POP", to reduce the size of the search space of the problem, and improve the performance of the GA by increasing the probability of reaching the global optimum. To implement this strategy, during the creation of the initial population, each time a new individual (i.e., test case) is generated, it is not constructed from all the available public methods and constructors of the CUT; instead, it is chosen based on a randomly constructed configuration.

The following steps are implemented to generate each configuration and to create the initial population automatically. First, all public methods and constructions of the

CUT are identified. Next, a random configuration, called C_r, is generated with only a set of randomly chosen public methods and constructors of the CUT. Finally, when generating a new individual, a sequence of method calls is generated from all possible sets of predefined available public methods and constructors in the generated C_r.

The rationale is that an individual generated using a randomly selected configuration, which leaves out some methods and constructors, has a higher chance of increasing the diversity of the initial population and avoiding the GA from premature convergence and being stuck in a local optima [68].

6.3 Seeding constants from source code

When the predicates of the CUT involve constants values and specific strings, randomly generating the right values to cover these branches can be challenging. One way of circumventing this problem is to collect constant values from the source code, and then enhance the search through seeding with these constant values [28]. For example, consider the following branch condition (if inputs.equals ("bar")). During the search (e.g., initial population and mutation operators), using the string value "bar" as a constant seed when generating string inputs may help the search to easily cover this branch [28].

There have been several seeding strategies used to improve different aspects of the search-based software testing (SBST) [4][5][28][35]. For example, Fraser and Arcuri study a seeding strategy that is applied when generating new or modifying existing test cases and show that the use of seeding can significantly improve the automated SBST tool EvoSuite [28]. In their work, a constant pool is populated with extracted concrete values from the source code during the instrumentation phase. Then, whenever attempting to randomly generate a new constant value, with a fixed probability $P_{Constant}$, a constant

value is randomly selected from the constant pool. They show that $P_{Constant} = 0.2$ gives the best results compared to other values (e.g., 0.4, 0.6, and 0.8).

As previous works have shown, seeding constant values extracted from source code of the class under test (CUT) can improve the performance of the search based test generation. However, all previous approaches use a constant seeding probability $P_{Constant}$ to randomly use one of the collected constants, rather than a random new value during test generation. Thus we make a few modifications to seed constant values collected from the source code of the CUT into the search process. We call this approach Directed-Seeding Values (**DSeed**). More details about **DSeed** are explained in the next section.

6.3.1 Guided-Seeding Strategy

Using the existing seeding approaches, there is a chance that seed collected constant values during the search do not cover their associated target branches. Fraser and Arcuri showed that depending on how the value $P_{Constant}$ is chosen, the seeding can be harmful and affect the coverage negatively in some classes [28]. The reason is that, the constant values collected are not specific to the predicates from which they were collected and they can aid only in covering branches that depend on these constants [4].

To mitigate this problem, we propose a Guided-Seeding strategy, **GSeed**, that directs the search to seed a constant value and focuses on methods relevant for calling the associated branch from which it was collected. For this purpose, we keep track of the methods and parameters in the test cases that have the potential to reach and execute uncovered branches, and we use this information to seed the constant values and to associated them with their respective inputs and branches [4]. Methods that do not contribute at all to trigger the associate branch are ignored altogether by the search [82].

More specifically, we statically analyze the predicate that is involved with the uncovered branch (i.e., target branch) and include any constant value if the predicate is dependent on a particular value and include a method that contains the target branch. We also create the following three 'method groups' formed using method dependence relations (MDR) [104][105].

Method under test consists of the methods that call directly or indirectly the target branch (i.e., associated branch).

Parameter provider consists of all the methods and constructors that return an object that can be passed as a parameter to the *methods under test*.

State changer consists of all the methods that may change the state of member fields in the *methods under test*.

Then, during the search, whenever attempting to generate a new constant value, with a certain probability, the constant value associated to the target branch is chosen rather than randomly generated. In addition, when randomly choosing a mutation operator (i.e., insert, update or delete), one of the method groups is selected at random and contains all methods that can influence the target branch, instead of exploring random of irrelevant method calls.

6.4 Empirical Evaluation

We now describe the evaluation of the effectiveness of our proposed strategies. Specifically, we assess how affective our proposed strategies are in improving the effectiveness of the GA by considering the following research questions:

1. **RQ1:** How does each of our population enhancement approaches (i.e, Directed-POP and SWARM-POP) affect branch coverage?

2. **RQ2:** What is the impact on branch coverage of using the guided-seeding strategy (GSeed) for seeding constants?

3. **RQ3:** How much higher percentage of branch coverage is achieved by Directed-POP, SWARM-POP, and GSeed compared to GAMDR and MAMDR, respectively?

The research questions were addressed using classes adopted from different open source code, as described in the next section.

6.4.1 Case Studies

To answer these research questions, we selected a set of 15 classes taken from the literature. More precisely, our selection criteria for classes were (a) represent cases where SBST (i.e., EvoSuite) struggle to achieve high coverage and (b) represent string problems in SBST.

An overview of the case studies selected for the experiments in this evaluation is shown in Table 6.1. They include fifteen (15) Java classes from 5 open source projects, and tend to perform different complex operations [74]. These case studies represent classes where different search-based techniques such as eToc [97] and EvoSuite [31] scored very low branch coverage as a reported in [1][36][74], and [85].

Project	Class	#Methods	NCSS	#Branches
Commons Codec	language.DoubleMetaphone	38	564	443
Conzilla	identity.PathURN	4	17	8
	identity.ResourceURL	5	22	10
	identity.URL	6	39	22
Joda-Time	format.DateTimeFormat	25	353	145
	format.DateTimeFormatter	45	256	88
lang3	lang3.ArrayUtils	233	1558	1104
	lang3.BooleanUtils	40	250	250
	lang3.ClassUtils	41	371	228
	lang3.JavaVersion	5	39	20
	math.NumberUtils	45	454	363
	time.DateUtils	58	425	227
NanoXML	nanoxml.CDATAReader	4	57	26
	nanoxml.ContentReader	4	68	26
	nanoxml.NonValidator	15	240	83

Table 6.1: Details of the test subjects used in the empirical study.

6.4.2 Experimental Set-up

To evaluate the effectiveness of our proposed strategies, we carried out two different sets of experiments, each one addressing one of each research questions. We ran each of enhancement strategy for 5 minutes on each test subject. All the experiments were repeated 30 times to take the randomness nature of the search based optimization into account. We ran GA with default random population on the case studies and measured its coverage. This measured coverage forms a baseline for comparing GA with and without the proposed enhancement strategies. Finally, all the experiments in the evaluations were conducted on a machine with Intel(R) Xeon(R) CPU E31240 V2 @ 3.40GHz and 14 GB RAM, running Red Hat with Kernel Linux 2.6.32.

6.4.3 RQ1: Optimizing the Initial Population

Class	GA	Directed-POP	SWARM-POP
language.DoubleMetaphone	76.80	**77.31**	76.00
identity.PathURN	47.92	**53.75**	51.25
identity.ResourceURL	67.33	75.33	76.00
identity.URL	59.09	68.18	68.79
format.DateTimeFormat	78.87	**81.10**	76.97
format.DateTimeFormatter	53.71	**58.79**	51.70
lang3.ArrayUtils	92.80	94.50	94.70
lang3.BooleanUtils	83.11	85.64	85.24
lang3.JavaVersion	46.33	53.33	**54.67**
math.NumberUtils	85.94	93.44	93.94
time.DateUtils	70.82	80.97	**82.35**
nanoxml.CDATAReader	71.15	**77.56**	76.15
nanoxml.ContentReader	70.90	**79.87**	74.36
nanoxml.NonValidator	23.09	**29.76**	20.04
Average	66.27	72.76	70.15

Table 6.2: Average Coverage Per Classes when Directed-POP and SWARM-POP initialization Strategies Are Employed.

In the first set of experiments, we addressed the first research question on whether our initial population enhancement approaches help increase branch coverage achieved by a simple GA approach [13][79]. For this purpose, we ran GA with and without the two different enhancement strategies for the initialization of the first population.

Table 6.2 shows the results obtained by the first experiment for **Directed-POP** and **SWARM-POP** initialization strategies. The number of covered branches increases with **Directed-POP** for all the test subjects, when the coverage is increased from 66.70% to 72.76%.

Covered branches also increase with **SWARM-POP** for all the test subjects except for language.DoubleMetaphone, format.DateTimeFormat and format.DateTimeFormatter

test subjects. The primary reason for lower coverage is that, the classes contain methods that require desired object states for the arguments. These desired objects help to cover the target branches in the methods. For example, the class format.DateTimeFormat contains 145 branches, 58 of which are in one private method called format.DateTimeFormatter(DateTimeFormatterBuilder builder, String pattern).

```
1.   public class DateTimeFormat {
2.    ...more public and private methods omitted here ...
3.      private void parsePatternTo(DateTimeFormatterBuilder
     builder, String pattern) {
4.          ...more statements omitted here ...
5.           switch (c) {
6.                case 'G': // era designator (text)
7.                   builder.appendEraText();
8.                   break;
9.                case 'C': // century of era (number)
10.                  builder.appendCenturyOfEra(tokenLen,
     tokenLen);
11.                  break;
12.               case 'x':  // weekyear (number)
13.                 ...4 more statements omitted here...
15.     }//end parsePatternTo method
14.   ...more public and private methods omitted here ...
15.  }
```

Figure 6.1: format.DateTimeFormat class

Figure 6.1 shows an example where target branches in a private method indirectly depend on a desired object state for the class DateTimeFormatterBuilder. The **SWARM-POP** fails to include all the require invoking method calls to instantiate the class DateTimeFormatterBuilder in the initial population.

Enhancing the initial population with a method dependence relation, **Directed-POP**, improves the branch coverage percentage compared to a GA for the DateTimeFor-

mat and DateTimeFormatter. In both classes, average coverage increases from 78.87% to 81.10% and from 53.71% to 58.79%, respectively. The better performance of **Directed-POP** can be explained as follows: When the initial population is entirely generated randomly, the chances of selecting a required DateTimeFormatterBuilder class are very small. On the hand, with the assistance of the static analysis, the **Directed-POP** identifies another method **DateTimeFormatterBuilder appendLiteral(char c)** in the DateTimeFormatterBuilder class, which returns an object of the DateTimeFormatterBuilder class. **Directed-POP** invokes **DateTimeFormatterBuilder appendLiteral(char c)** when calling DateTimeFormatter(DateTimeFormatterBuilder builder, String pattern) in some individuals that are able to generate sequences for creating objects of the the DateTimeFormatterBuilder class. Our results show that the static analysis phase improves the initial population and produces a much better starting individuals for the search to achieve higher branch coverage.

6.4.4 RQ2: Guided Seeding Constants

In the second set of experiments, we addressed the second research question which asks what the impact on branch coverage of using the guided-seeding strategy. To address this question, we compared the effectiveness of the guided seeding, **GA-GSeed**, to the approach proposed by Fraser and Arcuiri [28]. We called this approach unguided seeding, **GA-USeed**. To compare **GA-GSeed** with **GA-USeed**, we ran GA with the two different seeding strategies. To avoid a bias in our results, we set the value of $P_{Constant} = 0.2$ for **GA-USeed**, as it was the optimal probability value that gave the best results in the empirical work carried out by Fraser and Arcuri [28].

Table 6.3 shows the branch coverage achieved by GA, GA-USeed and GA-GSeed. In

Class	GA	GA-USeed	GA-GSeed
language.DoubleMetaphone	76.80	84.26	85.05
identity.PathURN	47.92	37.92	**65.00**
identity.ResourceURL	67.33	73.00	**82.67**
identity.URL	59.09	70.30	**78.79**
format.DateTimeFormat	78.87	81.17	**86.64**
format.DateTimeFormatter	53.71	49.70	**55.64**
lang3.ArrayUtils	92.80	95.19	**98.69**
lang3.BooleanUtils	83.11	86.92	86.88
lang3.JavaVersion	46.33	100.00	100.00
math.NumberUtils	85.94	94.02	94.61
time.DateUtils	70.82	82.85	82.91
nanoxml.CDATAReader	71.15	75.13	**94.74**
nanoxml.ContentReader	70.90	77.56	**90.90**
nanoxml.NonValidator	23.09	24.76	**38.29**
Average	66.27	73.77	81.48

Table 6.3: Results of the branch coverage achieved by Directed Seeding.

both seeding strategies, average coverage increases from 66.27% to 73.77% and 81.48%, respectively. As shown in our results, GA-GSeed outperforms GA-USeed by covering target branches that GA-USeed failed to cover.

As the Table 6.3 shows, the GA-GSeed strategy (which uses directed seeding) achieves significantly higher branch coverage than the GA-USeed strategy (which does not) on the classes PathURN, DateTimeFormat, ArrayUtils, CDATAReader, ContentReader and NonValidator classes. GA-GSeed achieves the same coverage for the rest of the classes with GA-USeed.

A closer look at the target branches that GA-USeed was not able to cover without the directed seeding illustrates where the current seeding strategy faces problems. Figure 6.2 shows an example where GA-USeed has problem covering target branches.

The target branches are contained in the constructor of the PathURN and dependent

```
1.  public class PathURN extends URN {
2.     public PathURN(String nuri) throws MalformedURIException {
3.        super(nuri);
4.        if(uri.length() == secondColonLocation + 1)
5.        throw new MalformedURIException("Empty Path URN", uri);
6.        if (uri.charAt(secondColonLocation + 1) != '/' )
7.        throw new MalformedURIException("Path URN has no ...);
13.        if(!uri.regionMatches(colonLocation + 1, "path", 0,
13.          secondColonLocation - colonLocation - 1))
14.        throw new MalformedURIException("The identifier was ...);
15.     }//end PathURN constructor
14.   ...more public methods omitted here ...
15.  }
```

Figure 6.2: identify.PathURN class

on a string value and can only be covered if GA-USeed finds the correct value for each target branch, such that an PathURN object is constructed. However, the chances of finding the correct values for calling a constructor can be very small. Without domain-specific knowledge, random seeding in GA-USeed many generated tests that do not trigger their target associated branches. The reason is that GA-USeed chooses the extracted values and methods to call randomly from all constant values and methods in the CUT, even though only a subset of all methods are used for triggering the associated target branches. However, the static analysis used in GA-GSeed helps to seed the extracted constant values into the search space when targeting their associated branches and guides the GA towards methods relevant for triggering their associated target branches. As a consequence, the related methods influence the random decisions of the GA-GSeed and leads to improved coverage of the PathURN class.

6.4.5 RQ3: Comparison with GAMDR and MAMDR

We next address the third research question regarding comparing branch coverage achieved by the proposed enhancement strategies with GAMDR and MAMDR (see Chapter 4 and 5 for more details).

Table 6.4 shows our results. The results show that the integration of MDR into the GA and MA improves the branch coverage much over the standard GA and MA. By focusing only on relevant methods, GAMDR, Directed-POP, MNS, and MWS improves the chance of generating desired sequences of method calls for each target branch and further improves the chance by covering all target branches.

In summary, these encouraging results suggest that information control dependencies plays an important role to improve code coverage by assisting search-based approaches. We can conclude that the integration of information control dependencies into the search-based techniques leads to an improvement with strong statistical significance.

6.5 Threats to Validity

We are aware of the following threats to the validity of our results. First, threats to *internal validity* might come from how the empirical study was conducted. To limit these threats, we carefully tested our testing framework to reduce the probability of having any faults. In addition, all the experminet were repeated 30 times to reduce any random aspects in the observed results. Second, threats to *construct validity* might exist because we gave the priority to the branch coverage to compare the performance of our techniques. This measurement does not take into account how difficult the produced test suites will be to manually evaluated. However, it is widely accepted that the higher the

Class	RT	EvoSuite	GA	GAMDR	Directed-POP	SWARM-POP	GA-USeed	GA-GSeed	MA	MNS	MWS
language.DoubleMetaphone	79.80	80.66	76.80	84.54	77.31	76.00	84.26	85.05	81.90	82.95	85.22
identity.PathURN	44.17	39.17	47.92	82.92	53.75	51.25	37.92	65.00	48.75	50.00	53.75
identity.ResourceURL	74.00	90.00	67.33	91.00	75.33	76.00	73.00	82.67	71.00	84.00	94.00
identity.URL	39.39	73.48	59.09	87.42	68.18	68.79	70.30	78.79	91.36	93.18	98.18
format.DateTimeFormat	80.92	68.69	78.87	84.90	81.10	76.97	81.17	87.38	81.15	83.75	89.06
format.DateTimeFormatter	47.77	69.47	53.71	59.58	58.79	51.70	49.70	55.64	81.48	84.77	84.32
lang3.ArrayUtils	99.09	61.42	92.80	99.18	94.50	94.70	95.19	98.69	94.23	94.28	95.92
lang3.BooleanUtils	84.80	79.99	83.11	89.40	85.64	85.24	86.92	86.88	84.36	85.44	87.40
lang3.JavaVersion	65.33	100.00	46.33	75.50	53.33	54.67	100.00	100.00	61.83	93.67	100.00
math.NumberUtils	92.07	79.94	85.94	94.07	93.44	93.94	94.02	94.61	90.57	94.07	96.23
time.DateUtils	69.27	89.00	70.82	82.95	80.97	82.35	82.85	82.91	80.18	82.82	84.58
nanoxml.CDATAReader	66.03	68.85	71.15	90.38	76.15	77.56	75.13	94.74	77.05	97.05	96.67
nanoxml.ContentReader	74.36	86.28	70.90	90.38	79.87	74.36	77.56	90.90	82.56	90.51	91.67
nanoxml.NonValidator	24.11	15.98	23.09	35.41	29.76	20.04	24.76	38.29	30.61	35.93	38.37

Table 6.4: Evaluation results showing higher branch coverage achieved by GA and MA with the assistance of MDR.

coverage is, the better test suites are [16]. Lastly, threats to *external validity* regarding the selection of classes we analyzed may not be representative for all Java classes. To avoid selection bias, we select all classes from different recent studies on string problem in search-based testing [36] [28] [74] and report the results for all of them.

6.6 Conclusion

Genetic algorithm (GA) shows a promising approach for the task of testing object oriented programs. The performance of the GA is dependent on a multitude of factors and parameters. initial population and Seeding constant are examples of factors that may strongly influence the performance of the GA. In this chapter, we proposed two new strategies to create an initial population, and analyzed a new different seeding technique. The aim is to enhance the performance of the GA and reduce the risk of being stuck in a local optimum. Our primary results show that beginning a search with specific individuals instead of randomly generating the initial population enables the improvement of the coverage and reaching the global optimum. In addition, instead of relying on randomly seeding constant, we include more domain specific information as a guide for the seeding constants. Using related methods, we have evidence that directed seeding leads to significantly improved the performance of GA and achieved better coverage.

Our initial set of experiments showed that the proposed enhancement techniques can lead to increased branch coverage. Our intention is to continue to accommodate other structural testing criteria, such as data-flow coverage or mutation coverage. Our future work also includes the enhancement of other aspects of the genetic algorithm particularly the representation of the fitness function to speed up the evaluation of the GA.

Chapter 7: Conclusion and Future Work

In this chapter we will summarize the achievements of this thesis and suggest possible future directions for research.

7.1 Summary of Achievements

In this thesis, we proposed different approaches that novelty rely on statically analyzing the internal structure of a class under test (CUT) to improve the effectiveness of search-based test data generation techniques. The proposed approaches are based on a genetic algorithm (GA) and a memetic algorithm (MA), which use search-based optimization to target all uncovered barnacles at the same time. Our approaches are novel in using information control dependencies from a static analysis to generate relevant sequences of method calls that achieve high code coverage. The results of empirical studies on several Java open-source projects showed that the statically-identified dependence information helped the GA and MA algorithms to achieve higher branch coverage than the current state of the art. The results represent a step towards a new direction on leveraging the information control dependencies to assist search-based test generation approaches to achieve higher code coverage.

The aims and objectives of this thesis were as follows:

1. To address the limitations of the existing search-based techniques for testing object-oriented programs

2. To develop new techniques to improve the capability and the performance of the search-based approaches for the automatic generation test data for object-oriented programs.

3. To perform an empirical investigation which evaluates the new techniques against the state of the art techniques.

The first of the objectives was addressed by comparing and evaluating the random testing and search-based testing techniques. We evaluated these techniques by comparing and contrasting their strengths and limitations in handling large size and high complexity programs. These limitations are the primary motivation that led to the proposal of this thesis.

After successfully pointing out that the exploration of the search space is the fundamental challenge facing the search-based approaches, the second objective was achieved by combining a static analysis in a novel way with a genetic algorithm (GA) in a tool called GAMDR [2]. The tool is driven by the method dependency relations (MDR) to optimize the search space exploration. Unlike previous search-based approaches, this combination enables the proposed approach to explore high complexity code in order to achieve high branch coverage.

In this thesis, we also make the contribution of enhancing the effectiveness of the GA. In particular, we extended the GA test generation into a Memetic Algorithm, by equipping it with several Local Search operators. These operators exploited MDR to allow the search for test cases to function more effectively [1]. In addition, to enhance the initial population of the GA, we leveraged the method dependence relations given by Zhang et al. [104] and SWARM approach given by Groce et al. [44] to provide the GA with a high-quality initial population that help the GA to achieve higher code coverage.

The strategy guided-Seeding constant values, also was proposed to increase the likelihood of covering target branches that rely on specific values. To this end, a static analysis was applied to direct the search to seed a constant value and focus only on methods relevant for calling the associated branch from which it was collected.

The last objective of the thesis was achieved by evaluating and comparing the proposed techniques to popular search algorithms. Different empirical studies involved a large size of classes taken from open source programs. The results show that the proposed approaches scale well to complex hard-to-cover programs and demonstrate a significant improvement in branch coverage over popular approaches.

In summary, Chapter 2 starts by describing the various features of object-oriented programming and how these features affect testing of object-oriented systems. The chapter continues with an explanation of the differences between procedural and object-oriented programs. Finally, the chapter provides background information on most of the topics covered in our proposed technical approaches.

Chapter 3 illustrates an extensive literature review of automated test input generation techniques. Particularly, we reviewed search-based testing and input domain reduction techniques.

Chapters 4, 5 and 6 propose approaches to improve the effectiveness of the search-based approaches. To this end, we leverage the method dependence relations to identify relevant methods and parameters that need to be mutated in order to cover particular branches. Unlike previous search-based approaches, the works in this thesis have a number of advantages. **1)** It focuses on the root cause of the failure to cover target branches. **2)** It focuses only on the relevant parts of the individuals (i.e., test cases) that affect the execution of the target branches. **3)** It implements a domain reduction mechanism to speed search space exploration. These strengths together enable the proposed

approaches to explore high complexity code in order to achieve high branch coverage.

7.2 Summary of Future Work

7.2.1 Future Work

This thesis has contributed effective techniques to significantly improve the performing of the search-based software testing to maximize branch coverage, particularly in the field of test input generation for object-oriented programs. However, the proposed approaches have a few limitations. This section discusses the limitations of the current implementation of our proposed approaches, and proposes a number of interesting research directions that have the potential for further investigation.

- Our proposed approaches focus exclusively on generating test suites that achieve as high as possible branch coverage. It is widely accepted that the higher the coverage is, the better test suites are [16], but this measurement does not take into account the difficulty faced by developers in evaluating the produced test suites, and adding test oracles in terms of assertions. Therefore, it would be interesting to enlarge the scope of the proposed techniques to automatically generate effective assertions and evaluate the fault detection capability of the generated test suites.

- We showed the ability of method dependence relations (MDR) to guide the genetic algorithm (GA) and the hill climbing (HC) to generate tests with a higher branch coverage than the pure GA and EvoSuite. However, it has been reported that search algorithms usually depend on several parameters, such as population size, mutation rate, and type of crossover [10]. It would be interesting to carry out a

large empirical analysis on different combinations of parameters and values, or use more advanced parameter control techniques [26] and show that the superiority of our proposed techniques hold.

- We combined MDR with the GA and the local search (HC), and evaluated our proposed techniques against the pure random testing, the pure GA, the HC, and EvoSuite. There is a potential opportunity to combine MDR with other search optimizations, such as Korle'Alternating Variable Method [64], and conduct further experiments to show whether or not such a combination further improves the achieved coverage.

- We showed that the exploitation of the domain knowledge can improve different aspects of the GA, such as the initial population. However, a study shows that generating desirable object instances has been a significant challenge for automated test generation tools [59]. Thus, it is interesting to move a step further and leverage the domain knowledge to alleviate such problems, and improve generating required instances for classes with complex internal states. For this purpose, one should investigate the capturing object instances from different search phases, and exploit these object instances to guide the search in generating test cases [59].

7.2.2 Final Thoughts

All the experiments in this thesis were performed on different Java projects representing instances of difficult problems from an search-based test software testing (SBST) approaches point of view, such that the integration of information control dependencies and SBST approaches leads to clear improvements [36]. However, not all real software

actually exhibit the necessary properties; data dependencies may be rare in real software. To demonstrate whether data dependencies are common in Java programs, we randomly chose and analyzed five different open source projects: `tinySQL`[1], `Barbecue`[2], `OMJState`[3], `JETT`[4], and `riak`[5]. Then, we randomly selected 10 classes from each project, and applied our static analysis tool to identify method dependence relations based on the fields they read or write. The results of our empirical analysis show that 10% of the classes exhibit the data dependencies information.

We next show different examples demonstrate the usefulness of the static analysis phase in practice and provide useful evidence to identify the set of relevant methods for each target branch.

The first example taken from the `tinySQL` project and shows the static analysis is effective in identifying the set of relevant methods from different classes that can be called to reach a target branch. Our static analysis identifies that the method `executeQuery` from the `tinySQLStatement` class requires the constructor of the `tinySQLDriver`, `tinySQLConnection`, and `tinySQLStatement`, classes to be called in a correct order. To achieve high code coverage of the `executeQuery`, a valid `tinySQLDriver`, a `tinySQLConnection`, and a `tinySQLStatement` object must be created before the method is called. Testing only relevant methods and constructors might help SBST approaches to create `tinySQLDriver`, `tinySQLConnection`, and `tinySQLStatement` objects in a correct order and generate more effective sequences of method calls.

The second example shows a case where a method parameters cannot influence the coverage of the target branches. The class `GraphicOutput` taken from `Barbecue`

[1]https://sourceforge.net/projects/tinysql
[2]http://barbecue.sourceforge.net/
[3]http://omjstate.sourceforge.net/
[4]http://jett.sourceforge.net/
[5]https://github.com/krestenkrab/riak-java-pb-client

project. The class contains 5 methods, and all the target branches depend on two private data members, `g` and `savedColour`. If a SBST approach fails to call the constructor `GraphicsOutput`, which changes the state of the member fields `g` and `savedColour`, the generated tests will fail to cover most of the target branches. However, the static analysis identifies that the method `beingDraw` and the constructor of the classes `GraphicsOutput`, `CenteredLabelLayout`, `MarginLabelLayout`, and `DefaultLabelLayout` are relevant and can influence the coverage of the target branches in the method `beingDraw`.

The classes `FormulaScanner` and `FormulaParser`, taken from `jett`, contain all the target branches only in one public method. For example, the class `FormulaScanner` has five private data members and six public methods. The method `Token getNextToken()` contains all the target branches, and does not have any parameters that help to cover these target branches. The static analysis can be used to identify related methods (i.e., `void reset()` and `void setFormulaText(String formulaText)`) based on the read/write data members and recommend to test them together. By excluding the irrelevant methods, we essentially reduce the search space for testing the target branches, and allow the SBST approach to better explore the search space, reaching more target branches; thus increasing code coverage.

The third example shows an example where a method parameter specifies an instance of Object, such that a SBST approach needs to find appropriate values for type parameter Object in order to cover all branches. The class `StringMatchesGuardCondition` and `IntegerGreaterThanGuardCondition`, which are taken from `OMJState` project, contains branches that rely on an object type. For example, to cover all the branches in the method `evaluate (Object o)` in the class `StringMatchesGuardCondition`, an instance of the `Event` class is required in a relevant state. The challenges in this

method is that a SBST approach needs to create an instance that matches the method signature when calling `evaluate`. However, our static analysis identifies that the method depends on `Event` class. By identifying this relation, a SBST approach can create a suitable `Event` object and can easily cover all dependent branches in the method `evaluate (Object o)`.

The final example shows some classes are hard to instantiate. The constructor of the classes `StateMachine` taken from `OMJState` project, and `BucketProperties` taken from `riak` project, require objects to generate the desired object state. For example, the class `StateMachine` has three private data members and two public methods. The constructor of the class modifies all the data members and contains a loop over `Transitions`. If a SBST approach fails to generate a desirable object instance required for the data member `Transitions`, an exception is thrown and no `StateMachine` object will be created. This class similar to the PathURN class in the Conzilla project, which was used in our evaluations. We expect that, with a SBST approach alone, it is difficult to obtain any coverage. However, the static analysis can identify relevant accessible constructors of the `StateMachine` class, and allows the SBST approach to concentrate on creating new instances of the `StateMachine` class more frequently.

We believe that when testing sizable and complex programs where the search space is too large to exhaustively explore, the ability of our proposed approaches' to reduce the search space and to precisely guide the SBST approaches by focusing on relevant methods and constructors for achieving high branch coverage.

Bibliography

[1] Ali Aburas and Alex Groce. An improved memetic algorithm with method dependence relations (mamdr). In *Quality Software (QSIC), 2014 14th International Conference on*, pages 11–20. IEEE, 2014.

[2] Ali Aburas and Alex Groce. A method dependence relations guided genetic algorithm. In *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, pages 267–273, 2016.

[3] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder K Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, 2010.

[4] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 3–12. IEEE Computer Society, 2011.

[5] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16(3):175–203, 2006.

[6] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[7] James H Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. Random test run length and effectiveness. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 19–28. IEEE Computer Society, 2008.

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

[9] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1–10. IEEE, 2011.

[10] Andrea Arcuri and Gordon Fraser. On parameter tuning in search based software engineering. In *Search Based Software Engineering*, pages 33–47. Springer, 2011.

[11] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 79–90. ACM, 2014.

[12] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random testing: Theoretical results and practical implications. *Software Engineering, IEEE Transactions on*, 38(2):258–277, 2012.

[13] Andrea Arcuri and Xin Yao. A memetic algorithm for test data generation of object-oriented software. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 2048–2055. IEEE, 2007.

[14] Andrea Arcuri and Xin Yao. On test data generation of object-oriented software. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 72–76. IEEE, 2007.

[15] Andrea Arcuri and Xin Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[16] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. Testful: an evolutionary test approach for java. In *Software testing, verification and validation (ICST), 2010 third international conference on*, pages 185–194. IEEE, 2010.

[17] Luciano Baresi and Matteo Miraz. Testful: automatic unit-test generation for java classes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 281–284. ACM, 2010.

[18] David Binkley and Mark Harman. Analysis and visualization of predicate dependence on formal parameters and global variables. *Software Engineering, IEEE Transactions on*, 30(11):715–735, 2004.

[19] Grady Booch. *Object oriented analysis & design with application*. Pearson Education India, 2006.

[20] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.

[21] Tsong Yueh Chen, TH Tse, and Yuen-Tak Yu. Proportional sampling strategy: a compendium and some insights. *Journal of Systems and Software*, 58(1):65–81, 2001.

[22] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 84–94. ACM, 2007.

[23] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 71–80. IEEE, 2008.

[24] John Clarke, Jose Javier Dolado, Mark Harman, Rob Hierons, Bryan Jones, Mary Lumkin, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, et al. Reformulating software engineering as a search problem. *IEE Proceedings-software*, 150(3):161–175, 2003.

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

[26] Agoston Endre Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 3(2), 1999.

[27] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):63–86, 1996.

[28] Gordon Fraser and Andrea Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 121–130. IEEE, 2012.

[29] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 178–188. IEEE Press, 2012.

[30] Gordon Fraser and Andrea Arcuri. Handling test length bloat. *Software Testing, Verification and Reliability*, 23(7):553–582, 2013.

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

[32] Gordon Fraser and Andrea Arcuri. Automated test generation for java generics. In *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering*, pages 185–198. Springer, 2014.

[33] Gordon Fraser, Andrea Arcuri, and Phil McMinn. Test suite generation with memetic algorithms. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1437–1444. ACM, 2013.

[34] Gordon Fraser, Andrea Arcuri, and Phil McMinn. A memetic algorithm for whole test suite generation. *Journal of Systems and Software*, 103:311–327, 2015.

[35] Gordon Fraser and Andreas Zeller. Exploiting common object usage in test case generation. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 80–89. IEEE, 2011.

[36] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 360–369. IEEE, 2013.

[37] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[38] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[39] Jamie S Gordon and Robert F Roggio. A comparison of software testing using the object-oriented paradigm and traditional testing. *Journal of Information Systems Applied Research*, 2014.

[40] Alex Groce. Coverage rewarded: Test input generation via adaptation-based programming. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 380–383. IEEE Computer Society, 2011.

[41] Alex Groce, Alan Fern, Joel Pinto, Thomas Bauer, Anahita Alipour, Martin Erwig, and Carlos Lopez. Lightweight automated testing with adaptation-based programming. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 161–170. IEEE, 2012.

[42] Alex Groce, Klaus Havelund, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.

[43] Alex Groce, Chaoqiang Zhang, Mohammed Amin Alipour, Eric Eide, Yang Chen, and John Regehr. Help, help, i'm being suppressed! the significance of suppressors in software testing. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 390–399. IEEE, 2013.

[44] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 78–88. ACM, 2012.

[45] Crina Grosan and Ajith Abraham. Hybrid evolutionary algorithms: methodologies, architectures, and reviews. In *Hybrid evolutionary algorithms*, pages 1–17. Springer, 2007.

[46] H Gross and Arjan Seesing. A genetic programming approach to automated test generation for object oriented software. Technical report, Delft University of Technology, Software Engineering Research Group, 2006.

[47] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*.

[48] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 155–164. ACM, 2007.

[49] Mark Harman, Lin Hu, Rob Hierons, Chris Fox, Sebastian Danicic, Joachim Wegener, Harmen Sthamer, and André Baresel. Evolutionary testing supported by slicing and transformation. In *Software Maintenance, 2002. Proceedings. International Conference on*, page 285. IEEE, 2002.

[50] Mark Harman, Fayezin Islam, Tao Xie, and Stefan Wappler. Automated test data generation for aspect-oriented programs. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 185–196. ACM, 2009.

[51] Mark Harman, William B Langdon, Yue Jia, David R White, Andrea Arcuri, and John A Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–14. ACM, 2012.

[52] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. 2009.

[53] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Software Engineering, IEEE Transactions on*, 36(2):226–247, 2010.

[54] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, pages 1–59. Springer, 2012.

[55] Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. 2013.

[56] Kobi Inkumsah and Tao Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 425–428. ACM, 2007.

[57] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 297–306. IEEE, 2008.

[58] David Jackson. Mutation as a diversity enhancing mechanism in genetic programming. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1371–1378. ACM, 2011.

[59] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K Chang. Ocat: object capture-based automated testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 159–170. ACM, 2010.

[60] Joseph Kempka, Phil McMinn, and Dirk Sudholt. A theoretical runtime and empirical analysis of different alternating variable searches for search-based testing. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1445–1452. ACM, 2013.

[61] Sujata Khatri and RS CHILLAR. Analysis of features affecting testing in object oriented systems. *ANALYSIS*, 3(2):17–21, 2011.

[62] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[63] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[64] Bogdan Korel. Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16(8):870–879, 1990.

[65] Anton Kotelyanskii and Gregory M Kapfhammer. Parameter tuning for search-based test-data generation revisited: Support for previous results. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 79–84. IEEE, 2014.

[66] Yee Leung, Yong Gao, and Zong-Ben Xu. Degree of population diversity-a perspective on premature convergence in genetic algorithms and its markov chain analysis. *IEEE Transactions on Neural Networks*, 8(5):1165–1176, 1997.

[67] Jan Malburg and Gordon Fraser. Combining search-based and constraint-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 436–439. IEEE Computer Society, 2011.

[68] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[69] Phil McMinn. Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pages 153–163. IEEE, 2011.

[70] Phil Mcminn. An identification of program factors that impact crossover performance in evolutionary test input generation for the branch coverage of c programs. *Information and Software Technology*, 55(1):153–172, 2013.

[71] Phil McMinn, Mark Harman, Kiran Lakhotia, Youssef Hassoun, and Joachim Wegener. Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. *Software Engineering, IEEE Transactions on*, 38(2):453–477, 2012.

[72] Phil McMinn and Mike Holcombe. The state problem for evolutionary testing. In *Genetic and Evolutionary ComputationGECCO 2003*, pages 2488–2498. Springer, 2003.

[73] Phil McMinn and Mike Holcombe. Evolutionary testing using an extended chaining approach. *Evolutionary Computation*, 14(1):41–64, 2006.

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

[75] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, (3):223–226, 1976.

[76] Matteo Miraz, Pier Luca Lanzi, and Luciano Baresi. Improving evolutionary testing by means of efficiency enhancement techniques. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

[77] Pablo Moscato et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

[78] Alessandro Orso and Gregg Rothermel. Software testing: a research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, pages 117–132. ACM, 2014.

[79] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 75–84. IEEE, 2007.

[80] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Adoption and use of java generics. *Empirical Software Engineering*, 18(6):1047–1089, 2013.

[81] Yury Pavlov and Gordon Fraser. Semi-automatic search-based test generation. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 777–784. IEEE, 2012.

[82] Michael Pradel and Thomas R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *International Conference on Software Engineering (ICSE)*, 2012.

[83] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: An approach for debugging evolving programs. *ACM Transactions on Software Engineering and Methodology*, 2(3):1–0, 2001.

[84] José Carlos Bregieiro Ribeiro, Mário Alberto Zenha-Rela, and Francisco Fernández de Vega. Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. *Information and Software Technology*, 51:1534–1548, 2009.

[85] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *Software Engineering, IEEE Transactions on*, 41(3):294–313, 2015.

[86] Alexandru Salcianu and Martin Rinard. A combined pointer and purity analysis for java programs. 2004.

[87] Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for java programs. In *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385, page 199. Springer Science & Business Media, 2005.

[88] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 218–227. IEEE, 2008.

[89] Arjan Seesing. Evotest: Test case generation using genetic programming and software analysis. *Operations Research*, 2:393–410, 1954.

[90] Koushik Sen, Darko Marinov, and Gul Agha. *CUTE: a concolic unit testing engine for C*, volume 30. ACM, 2005.

[91] Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn. Random or genetic algorithm search for object-oriented test suite generation? In *Genetic and Evolutionary Computation Conference (GECCO 2015)*, pages 1367–1374. ACM, 2015.

[92] Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. Testing container classes: Random or systematic? In *Fundamental Approaches to Software Engineering*, pages 262–277. Springer, 2011.

[93] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 193–202. ACM, 2009.

[94] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. In *ACM SIGPLAN Notices*, volume 46, pages 189–206. ACM, 2011.

[95] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. Springer, 2008.

[96] Vedat Toğan and Ayşe T Daloğlu. An improved genetic algorithm with initial population strategy and self-adaptive member grouping. *Computers & Structures*, 86(11):1204–1218, 2008.

[97] Paolo Tonella. Evolutionary testing of classes. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 119–128. ACM, 2004.

[98] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 285–288. IEEE, 1998.

[99] Willem Visser, Corina S Psreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[100] Stefan Wappler and Frank Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1053–1060. ACM, 2005.

[101] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[102] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[103] Matt Weisfeld. The object-oriented thought process. 2009.

[104] Sai Zhang, Yingyi Bu, Xiao Sophia, Wang Michael, and D Ernst. Dependence-guided random test generation. In *University of Washington*. Citeseer, 2010.

[105] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 353–363. ACM, 2011.

[106] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.