University of Salerno

Department of Computer Science

Master of Science in Computer Science

Implementation and comparison of novel techniques for automated search based test data generation

Thesis in Software Engineering

Supervisor

Prof. Andrea De Lucia

Dario Di Nucci

Candidate

Giovanni Grano

Matr: 0522500255

Academic year 2014/2015

University of Salerno

Department of Computer Science

Master of Science in Computer Science

Implementation and comparison of novel techniques for automated search based test data generation

Thesis in Software Engineering

Supervisor

Prof. Andrea De Lucia

Dario Di Nucci

Candidate

Giovanni Grano

Matr: 0522500255

Academic year 2014/2015

For the past 33 years, I have looked in the mirror every morning and asked myself: "If today were the last day of my life, would I want to do what I am about to do today?" and whenever the answer has been "No" for too many days in a row, I know I need to change something

Steve Jobs

Contents

Chapter 1

Introduction

Testing is the process of finding differences between the expected behaviour specified by system models and the observed behaviour of the system [6]. The goal of testing is to design tests that exercise defects in the system to reveal problems. Unfortunately, it is impossible to completely test a non-trivial system essentially because testing is not a decidable problem and it needs to be performed under time and budget constraints.

A distinction between concepts of failure, error and fault should be done in order to clarify the concepts which will be discussed in this work. Firstly, a *failure* is any deviation of the observed behaviour from the specified behaviour. An *error* means the system is in a state such that further processing by the system will lead to a failure, which obviously causes anomalies in system intended behaviour. At last, a *fault* (also called defect or bug) is the mechanical or algorithmic cause of an error. The final goal of all testing activities is to maximize the number of discovered faults in order to increase the reliability of a system, namely a measure of success with which the observed behaviour conforms to the specification of its behaviour.

Software testing can be very costly and it is often accounted for more than 50% of total development costs. Thus, it is necessary to reduce its cost and improve its effectiveness by automating the testing process. Test data generation is the process of identifying program input data which satisfy selected

testing criterion and a test data generator is a tool which can automatically generate test data for a program. Among many testing activities, test case generation is one of the most challenging and critical tasks. For this reason, a good number of test case generation approaches has been advanced and investigated intensively.

Another common challenging problem is the reduction of the effort required for regression testing. This situation clearly requires a technique for an intelligent selection of test cases and for removing the redundant ones (test suite minimization). All these problems can all be naturally formulated as optimization problems to which search based optimization can be applied.

There are basically two testing methodologies:

- white-box (or structural) testing;
- black-box (or functional) testing.

With black-box testing, test cases are determined from the specification of the program under test, whereas with the white-box testing, also called *glass testing* or *open box testing*, they are derived from the program internal structure.

These techniques are used with the aim to generate a *test suite*, composed by a set of test case with the purpose of causing the largest number of failure in order to detecting faults. A *test case* is a set of conditions under which a tester will determine whether a software system or one of its components is working as it was described in its originally expected behaviour. The mechanisms for determining whether a software program has failed or passed a test is known as *oracle*. Its usage involves a comparisons of the output of the system under test, for a given test case, with the output that determined by the oracle.

1.1 Context

This work is placed in the context of structural testing. As said before, in software engineering there are many expensive activities performed during development and maintenance processes which should be automated in order to reduce their cost. Search Based Software Engineering (SBSE) attempts to reformulates these tasks as search problems [26, 28]. Since testing is one of the most expensive activities, SBSE is widely applicable to all software testing processes. In particular, regarding automated structural test data generation, the literature suggests two different techniques. Static structural test data generation is based on the analysis of the internal structure of the program and it does not require its execution. In the other hand, symbolic execution [32, 33] is a common example of static technique, but it presents several problems. Dynamic approaches, which requires the code execution, circumvent many of the problems associated with static techniques.

Despite the problem of automatically generating test inputs has been investigated for decades, the problem of automating the *test oracle* has received significantly less attention. Proposed techniques, including modelling, specifications or contract-driven development are not completely adequate, so in real cases the system behaviour must be checked by an human. Obviously, this current open problem represents a significant bottleneck that inhibits greater testing automation.

One simple and obvious way to reduce the oracle cost consists of reducing the size of the test suite produces as a result of automated test data generation. Obviously, this challenge contrasts with the goal to achieve the maximum coverage. Therefore, the problem of maximizing the coverage in the system under test, while reducing the resulting test suite size can be formulated as an optimization problem. With this aim, several approaches based on search-based techniques have been proposed in literature.

In this work we developed OCELOT, a tool for automated search based test case generation for C language. It implements an efficient approach,

based on McCabe's linearly independent paths, which combines the achievement of a good level of coverage with a reduced test suite size, in comparison with the standard single target approach.

1.2 Aim and objectives

In order to reduce the cost related to testing activities, it is possible to use some techniques with the aim to automate the testing process. The development of test data is a relevant problem which has been thoroughly investigated. However, the *oracle cost problem* remains comparatively less well-solved, so it requires a huge human intervention that limits the automation of the entire testing process.

Several methodologies and tools have been developed with the aim to address the problem of automated test case generation and the *oracle problem*, mainly trying to reduce the size of the resulting test suite. This work led to the implementation of a tool for structural search based test data generation, focused on code written in C language. This tool, OCELOT, introduces a new approach based on McCabe's linearly independent paths. This methodology reaches good results in terms of coverage and test suite size, compared against a random testing approach. However, it suffers of some problems such as the path infeasibility.

This works aim to investigate the recent literature in order to identify the most promising approaches for target selection in order to introduce them in the tool. So, the final aim of this work is the introduction of latest methodologies in OCELOT, in order to improve its effectiveness and efficiency.

An empirical study has been performed with the aim to evaluate the boost of effectiveness and efficiency introduced by these approaches, in comparison with the standard OCELOT one, based on basis paths. This study, in particular, compares the three methodologies in terms of coverage, test suite size and convergence rate.

1.3 Achievements

The main result of this work is the introduction and implementation of two recent methodologies for automated structural test data generation in OCELOT, a Java tool for search based test data generation.

Two approaches have been identified: the first, called MOSA (Many Object Sorting Algorithm) [47], reformulates the branch coverage problem as a many-objective optimization one. The second methodology [27], focuses on a control dependence graph analysis, in order to maximize the amount of structural collateral coverage achieved targeting a single branch.

The efficiency of these recent approaches have been evaluated through an empirical study, comparing them with the OCELOT current methodology based on the basis paths, introduced by Scalabrino in his master thesis work [51]. This study compares the three approaches in terms of coverage, test suite size and convergence rate. Regarding the first metric, both new algorithms do not significantly improve the performance of the basis paths approach. However, the analysis of result which refers to resulting test suite size and convergence rate denotes remarkable improvements. Regarding the resulting test suite size, both the newly introduced algorithms generate a test suite which is about the 50% smaller than one generated with the basis path approach. As mentioned, the reduction of the number of test cases is the simplest way to address the *oracle cost problem*, so in that extent, the tool clearly improves its efficiency. At last, in terms of convergence rate, the results shows that the many-objective algorithms outperforms its competitors.

1.4 Overview of the thesis

After this introduction, chapter 2 surveys the literature in the field of testing, in particular for search-based testing. It describes principal search techniques like genetic algorithm or hill climbing. There will be an overview of different techniques applied to evolutionary testing for structural test data generation, categorized on the basis of objective function construction.

Chapter 3 presents two recent methodologies available in literature, im-

plemented in the presented tool. The first approach proposes to reformulate the branch coverage criterion as a many-objective optimization problem. The second technique focuses on control dependence graph analysis in order to maximize the amount of structural collateral coverage.

Chapter 4 presents OCELOT, a structural test data generation tool for C language. The tool is build on top of CDT framework and implements different search based techniques. There will be presented the tool architecture with its execution process.

Chapter 5 presents an empirical study applying OCELOT to five different open source functions, with a complete description of the study and the obtained results.

Finally, chapter 6 closes the main body of the thesis with concluding comments and proposals for future work.

Chapter 2

Literature Review

This chapter introduce the field of Search Based Software Engineering, describing in particular the Search Bases Software Testing sub-area. A section reviews work in the field of search-based automatic test data generation. It focuses on the two most commonly used algorithms in structural search-based testing, namely genetic algorithms and hill climbing. Particular attention is paid to the global search method of genetic algorithms that forms the basis for most of the work in this thesis. This is followed by a discussion of techniques used for structural testing.

2.1 Search-Based Software Engineering

Search-Based Software Engineering (SBSE) is the name given to a body of work in which search based optimization is applied to Software Engineering. This approach has proved to be very successful and generic. SBSE reformulates the traditional tasks of software engineering as search problems [26, 28]. A search problem is one in which optimal or near optimal solutions are sought in a search space of candidate solution, guided by a fitness function that distinguishes between better and worse solutions. The first work about the application of search-based techniques to software engineering was proposed by Miller and Spooner in 1976 [45] who worked in software testing area, and later by Chang [7] who promoted the application of evolutionary algorithms

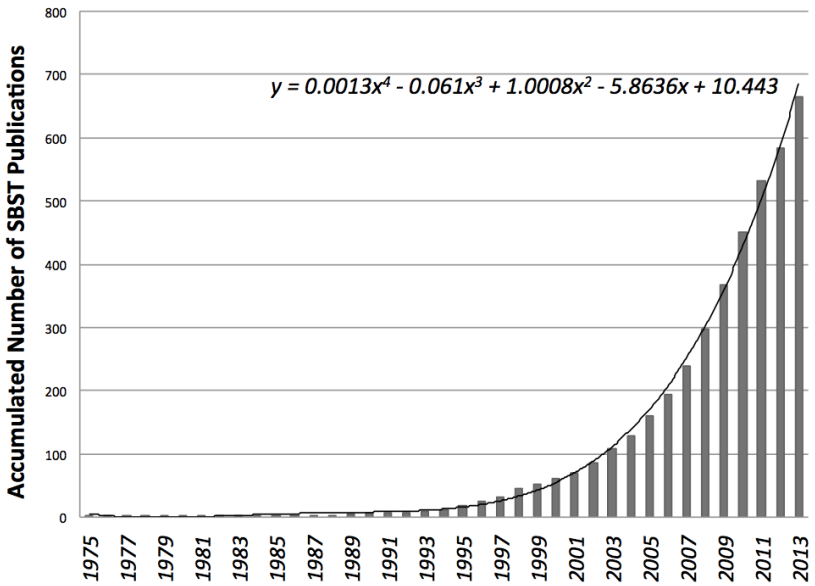


Figure 2.1: Cumulative number of SBST papers. The trend suggest a polynomial yearly rise in number of papers

for software management problems in 1994. The term SBSE was coined by Harman and Jones in 2001 [26]. Since then there has been an explosion of activity in this area, with SBSE being increasingly applied to a wide range of diverse areas within software engineering, some of which are already sufficiently mature to warrant their own surveys. For example there are surveys covering SBSE for requirements [63], design [46] and testing [40, 2] as well as general surveys of the whole field of SBSE [9, 23].

Search Based Software Testing (SBST) is the sub-area of Search Based Software Engineering concerned with software testing [25]. Testing is an expensive activity and a lot of problem in this field can be described as optimization problem, so SBSE clearly is widely applicable to all software testing processes. Test objectives find natural counterparts, as the fitness function, thereby facilitating SBSE formulation of many testing problems. As result, SBST has proved to be a widely applicable and effective way of generating test data, and optimising the testing process. Approximatively half the budget spent on software projects is spent on testing. So it is not surprising

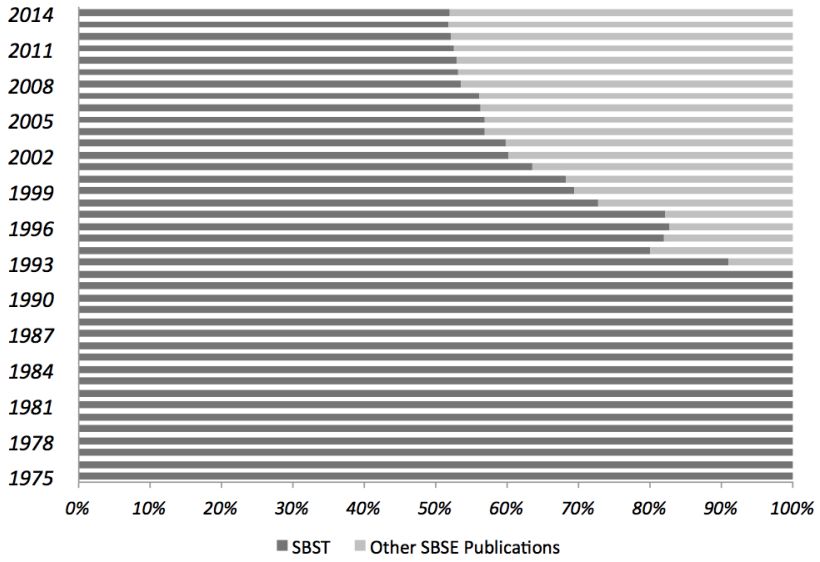


Figure 2.2: Changing ratio of SBSE papers that are SBST papers

that a similar proportion of papers in software engineering literature concern software testing. In fact, approximately half of all SBSE papers are SBST papers [28]. Figure 2.1 shows the growth in papers published in SBST. Data are taken from SBSE repository [62]. The figure clearly shows that the number of SBST papers is rising polynomially (if the trend continues, there will be more than 1,700 SBST papers before the end of this decade) indicating continued growth and interest in this approach to software testing problems.

In order to understand the contribution of software testing to the overall literature on SBSE, it is possible to observe the graph reported in Figure 2.2. This graph shows the relative proportion of SBST papers in the overall field of SBSE. As observable, the proportion of papers that concerns testing problems is reducing, although (as seen in Figure 2.1) their number is increasing. That reflects a huge growing interest in SBSE area, rather than decreasing interest in SBST. It is now sufficiently mature that it has transitioned from laboratory study to industrial application, for example at Daimler [58], Ericsson [1] and Microsoft [52].

2.2 Metaheuristic search techniques

Many results in the literature indicate that meta-heuristics are the state-of-the-art techniques for problems for which there is not an efficient algorithm. Meta-heuristics approximation approaches allow to solve complex optimization problems but they cannot guarantee that the solution found is the best or indeed it is the global optimal solution for the problem.

The use of meta-heuristic search techniques for the automated test data generation has been a burgeoning interest in recent years. Previous attempts to automate test generation process have basically been limited by the fact that this is an undecidable problem. In order to adapt meta-heuristic techniques to a specific problem, different decisions have to be made. The most important one is the choice of the fitness function; other important decisions involve, for example, the solution encodings or the choice of the best operators.

This section presents some meta-heuristic techniques that have been successfully used in software test data generation, categorized into local and global search techniques.

2.2.1 Local search

Local search is a meta-heuristic method for solving computationally hard optimization problems. Local search can be used on problems that can be formulated in the following way: find a solution maximizing (or minimizing) a criterion among a number of candidate solutions. These algorithms move from solution to solution into the search space by applying local changes, until an optimal solution is found or a time bound is elapsed.

Hill climbing

Hill climbing is a well known local search algorithm. It improves a solution starting with a randomly chosen one from the search space. Then the neighbourhood of this solution is investigated. If a better solution is found, it become the new current solution. This process is repeated until no improved

Algorithm 1: Description of a hill climbing algorithm

Select a starting solution $s \in S$;
repeat
  Select $s' \in N(s)$ such that $obj(s') > obj(s)$ according to ascent strategy;
  $s \leftarrow s'$;
until $obj(s) \geq obj(s')$, $\forall s' \in N(S)$;

neighbours can be found. A higher level description of basic hill climbing can be seen in algorithm 1.

The progressive improvement of hill climbing is compared to the climbing of hills in the *landscape* of a maximizing objective function. In this scenario, peaks characterize solutions with locally optimal objective values. *Hill climbing* is good for finding local optimum, i.e. a solution that cannot be improved by considering a neighbouring configuration, but it does not ensures to find the best possible solution, a.k.a. global optimum, in the entire search space. In such cases, the search becomes trapped at the peak of a hill, unable to explore other areas of the search space.

The relative simplicity of the algorithm makes it a popular first choice amongst optimizing algorithms. Moreover in literature we can find some variants to avoid the problem of local optimum, such as stochastic hill climbing or random-restart hill climbing [50]. The first of them does not examine all neighbours before deciding how to move, but it selects a random neighbour and decides whether to move to that neighbour or to examine another one; the second instead, avoid the problem of local optimum restarting the entire algorithm with randomly chosen starting points.

Simulated Annealing

Simulated annealing is a meta-heuristic first proposed by Kirkpatrick et al. [34]. This technique is similar to *hill climbing*, however, it attempts to avoid the problem of local optima in a different way. Its name originates from the analogy of the technique with the chemical process of annealing, the cooling of a material in a heat bath. In fact, this technique simulates that

Algorithm 2: High level description of a simulated annealing algorithm

```
Select a starting solution s ∈ S;
Select an initial temperature t > 0;
repeat
    it ← 0 repeat
        Select s' ∈ N(s) at random;
        Δe ← fit(s) − fit(s');
        if Δ < 0 then
            s ← s';
        else
            Generate random number r, 0 ≤ r < 1;
            If r < e^(−δ/t) Then s ← s';
        end
        it ← it + 1;
    until it = num_solns;
    Decrease t according to cooling schedule;
until stopping condition reached;
```

effect with a probability of accepting worse solution that depends from a control parameter known as the *temperature*. This concept is a fundamental property of simulated annealing because it allows for a more expensive search for the optimal solution. Initially the temperature is high and free movement is encouraged through the search space. With the progressive temperature reduction, the moves towards poorer solutions are less frequent. When a *freezing point* is reached the search behaves identically to *hill climbing*.

A high level description of a simulated annealing algorithm can be observed in Algorithm 2.

2.2.2 Global search

Hill climbing and *simulated annealing* operate with one candidate solution by one. On the other hand, there are some techniques which focus the entire search space at once, offering more robustness to local optima problem. Evolutionary algorithms [4] use the simulated evolution as a search strategy to evolve candidate solutions, utilizing some operators inspired by genetics

and natural selection.

Genetic algorithm are probably the most well known form of evolutionary algorithms. They have been introduced for the first time by John Holland during the late 1960s [22].

Genetic algorithms

Genetic algorithm are inspired by Darwinian evolution. In keeping with this analogy, each candidate solution is represented as a vector of component referred to *chromosomes*. The components of the solution are referred to as *genes* and all the possible values for each component are called *alleles*. Typically a genetic algorithm uses a binary representation, namely candidate solutions are encoded as strings of 1 and 0. For example, a vector of three integer $\langle 112, 255, 52 \rangle$ might be represented as $\langle 01110000, 11111111, 00110100 \rangle$. However, more natural representations to the problem may also be used, for example a list of floating point values.

Genetic algorithms maintain a population of solutions rather than just one current solution. Therefore the search is afforded many starting points and the chance to sample more of the search space than local searches. Population is iteratively recombined and mutated to evolve successive populations, known as *generations*. The main loop of a genetic algorithm can be see in Algorithm 3. The first generation is made up of randomly selected chromosomes. Population may also be *seeded* with selected individuals representing some specific domain information about the problem. This mechanisms may increase the chances of a fast search convergence. Each individual in the population is then evaluated for fitness.

On the basis of fitness evaluation, certain individuals are selected to go forward to the following stages of crossover, mutation and reinsertion into next generation. Various selection mechanisms can be used to select individuals which will be used to create offspring. In this scenario, the concept of *fitness* of individuals is crucial. The fitness of a candidate represents a measure that distinguishes between better and worse solutions and can be obtained directly from an objective function. The basic idea is to favour fit-

ter individuals in order to generate fitter offspring. However, too strong bias towards best individuals will reduce diversity in whole population, increasing the chance of premature convergence on a set of locally optimal solutions. This phenomenon is called *genetic drift* [49].

Holland's original algorithm [29] uses *fitness-proportionate selection*, where the expected number of times an individual is selected for reproduction is proportionate to the individual's fitness, in comparison with the rest of the population. This process is analogous to the use of a roulette wheel. *fitness-proportionate selection* has difficulties in maintaining the a constant *selective pressure* throughout the search. *Selective pressure* is the probability of the best individual being selective compared with the average probability if selection of all individuals. In *fitness-proportionate selection* the selective pressure remains usually too high and this can lead to a premature convergence. Linear ranking [59] and tournament selection [21] have been proposed to circumvent the problem.

Once the set of parents has been selected, recombination can be performed to create the new generation. Crossover is applied to individuals selected with a probability p_c. If crossover take place, the offspring are inserted into next generation, otherwise the parents are simply copied into. Different choices of operator are available such as *one-point* crossover, with splices two parents at randomly chosen position in the string to form two offspring; other operators may recombine using multiple crossover points.

Finally, elements of the newly-created chromosomes are randomly mu-

Algorithm 3: High level description of a genetic algorithm

```
Randomly generate or seed initial population P;
repeat
    Evaluate fitness of each individual in P;
    Select parents from P according to selection mechanism;
    Recombine parents to form new offspring;
    Construct new population P' from parents and offspring;
    Mutate P';
    P ← P';
until stopping condition reached;
```

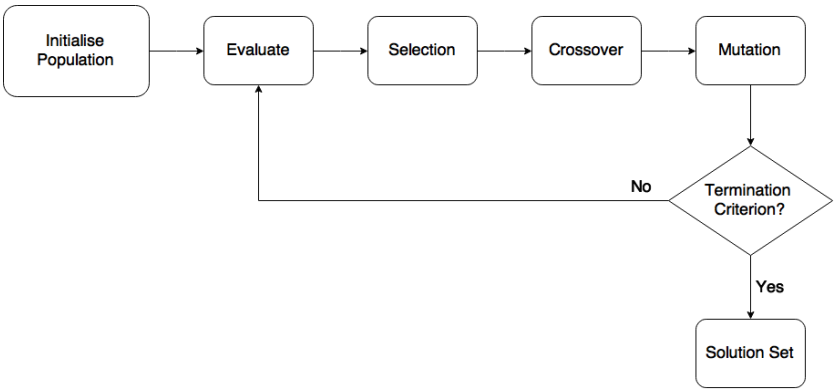


Figure 2.3: Genetic algorithm cycle

tated with the aim of diversifying the search into new areas of the search space. Typical process of evolution performed by a genetic algorithm previously described can be seen in Figure 2.3 at page 18.

Main decisions in genetics involves the representation of the solutions and the evaluation method used to compute the fitness function. Other decisions refers population size, crossover and mutation probability. Genetic algorithms success depends on their simplicity and, at the same time, on their ability to quickly find good solutions in large search space problems.

2.3 Search-based software test data generation

Structural testing is the process of deriving tests from the internal structure of code. Next sections summarize some approaches used in automating structural test data generation through meta-heuristic techniques.

The term *evolutionary testing* is used in literature to indicate the application of evolutionary algorithms to test data generation. As seen in Figure 2.4, all the different techniques can be categorized on the basis of objective function construction.

In *coverage-oriented approaches* an individual is rewarded on the basis of covered program structure. In this field, the work of Watkins [56] attempts to obtain full path coverage with the usage of an objective function

which penalizes individual that follow already covered paths. In general, all coverage-oriented approaches denote a lack of guidance for structures which can be executed only through a small portion of the input domain.

This problem is avoided in *structured-oriented approaches* which perform a separate search for each uncovered structure. These techniques can be categorized in *control-oriented*, *branch-distance oriented*, or *combined* approaches.

2.3.1 Control-oriented approaches

In *control-oriented* approaches, the objective function considers the branching nodes that need to be executed in some desired way in order to bring about execution of the desired structure [40]. In the approach of Jones et al. [30] to loop testing, the objective function is computed as the difference between the actual and desired number of iterations. Indeed, Pargas et al [48] use the control dependence graph of the function under test in order to address the statement and branch coverage problem. The sequence of control-dependent nodes is identified for each-structure. These are the branching nodes that must be executed in order to reach the structure. So, the objective function is simply calculated counting the number of control-dependent nodes executed. In Korel's work [35], a branch leading away from

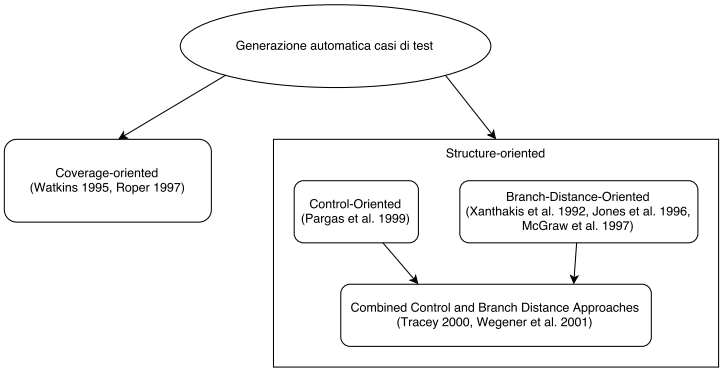


Figure 2.4: Classification of dynamic structural test data generation techniques using evolutionary algorithms

Relational Predicate	**Objective function *obj***
boolean	if $TRUE$ then 0 else K
$a = b$	if $abs(a-b) = 0$ then 0 else $abs(a-b) + K$
$a \neq b$	if $abs(a-b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a-b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a-b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b-a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b-a) + K$
$\neg a$	negation is propagated over a

Table 2.1: Tracey's objective functions for relational predicates

the target is called a *critical branch*, so the measure indicated by Pargas et is equivalent to the number of critical branches avoided by the candidate.

The main problem with control-oriented approach is the lack of guidance due to the plateaux generated by the objective function. Since no information is exploited from branch predicate, the objective function gives no help to change the flow of execution towards control dependent nodes essentially because all input data which executes the same nodes are assessed in the same way by objective function. Along these horizontal planes, the search becomes random. Pargas et al. proposed also a minimizing version of their objective function that can be computed as (*dependent* - *executed*) however, this solution does not solve the plateaux problem.

2.3.2 Branch-distance-oriented approaches

Branch-distance-oriented approaches exploit information from branch predicates. In general, for each condition node in object under test, the branch distance function takes an input set of values and evaluates the objective function on this values. Obviously this approach works very well with numeric data. Other kind of data, such as strings, needs different, and more complex, branch distance functions [42].

In the work of Xanthakis el al. [60], genetic algorithms are used to generate test data for nodes not covered by a random search. In this approach, a tester needs to select a path; then relevant predicated are extracted from it. The objective function is simply the sum of all branch distance values. Jones et al. [30] avoid the problem of the manual choice of the path. The objective

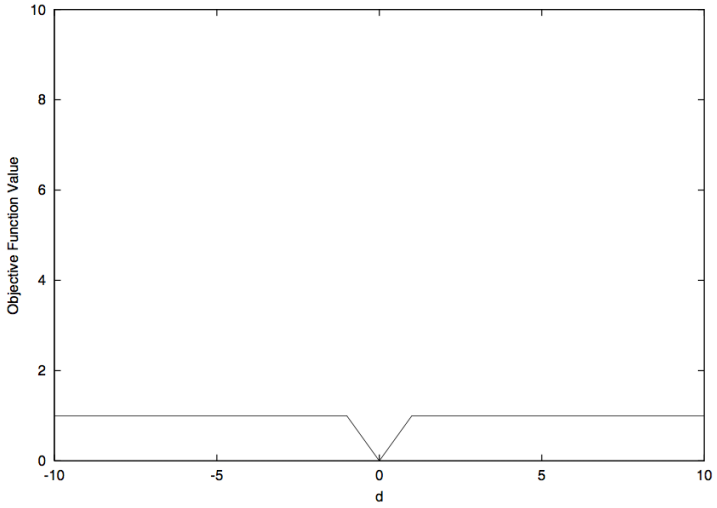


Figure 2.5: Objective function landscape for flag example

function is formed by the branch distance of the target branch. However, this approach does not provide a guidance to reach the desired target in the program structure. McGraw et al. [43] alleviate this problem by delaying an attempt to satisfy a branch condition until the node in question is reached by other individuals.

Most used objective functions in literature are proposed by Tracey et al. [54, 55]. These functions are inspired to original objective functions for relational predicate proposed by Korel [35]. Table 2.1 shows them.

Problems for objective functions

Although global search techniques are more robust than local about plateaux and local optima, they do not completely solve the problem. In particular, plateaux can be induced through the use of a *flag* variables in predicate branches. A flag is simply a boolean variable. The objective function for a predicates with a flag, consist only in two different values, one for the value true and one for the value false. In this situation the objective function does not provide any guidance and the evolutionary search performs as the same way than a random one. Figure 2.5 shows an example of objective function landscape caused by the flag problem.

Connective	Objective function *obj*
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$

Table 2.2: Tracey's objective functions for logical connectives

Bottacci [5] proposes a solution in which the predicate used for the distance calculation is substituted by the predicate used in assigning the flag value. Harman et al. [24] indeed suggest to transform the program removing flag variables replacing them with the expression that led to their determination.

2.3.3 Mixed approaches

Combined approaches make use of both branch distance and control information in order to build the objective function. Tracey's [53] work first identifies control-dependent nodes for the target structure, then performs a distance calculation when an individual takes a critical branch, using the functions showed in Table 2.1 for relational predicates and functions in Table 2.2 for *and* and *or* logical connectives. Then, the number of successfully executed control-dependent nodes is used to scale branches distance values. Wegener et al. [57] indeed map branch distance logarithmically into the range [0,1].

Different objective functions for different coverage criteria

The aim of applying evolutionary approaches to structural testing is the generation of a quantity of test data, leading to the best possible coverage of the respective structural test criterion. These criteria are divided into four categories and for each category exists; although each of them has a different definition of objective function, the basic form is always the following:

approach level + branch distance

The four criteria are:

- node-oriented criteria;
- path-oriented criteria;

- node-path-oriented criteria;
- node-node-oriented criteria.

Node-oriented criteria aim to cover specific nodes of the *control flow graph*. The approach level is calculated simply counting the number of target control-dependent nodes which have not been executed. Branch distance is calculated on predicate of alternative branch when the execution flow diverges in a critical branch.

Path-oriented criteria require the execution of specific paths through the *control flow graph*. Approach level is calculated on the basis of the length of identical initial path section. Branch distance calculation indeed is simply an accumulation of distance calculations made at each point of divergence from the intended path.

Node-path-oriented criteria require the achievement of a specific node and, the execution of a specific path through the control flow graph, starting from that node. The objective function is a combination of node-oriented and path-oriented calculation. The individuals which do not reach the initial node are treated as the same way described for node-oriented criteria.

Finally *node-node-oriented* criteria aim to execute program paths that cover certain node combinations on the control flow graph in a pre-determined sequence, without specifying a concrete path between nodes. The objectives function is a cumulative node-oriented strategy.

2.3.4 Normalized functions

For structural criteria, different heuristic have been designed to help the search. The most common heuristic is a linear combination of *approach level* and *branch distance*. As said, test data generation requires a fitness function f to evaluate test cases and usually this value should be minimized. If the target is covered during the execution of test case t, the function value reaches the value 0 and the search ends.

It is widely recognized that the approach level has more informative content than branch distance. However, for some input vectors, the branch

distance reaches very hight values, and this phenomenon alters the descriptive relation between approach level and branch distance. For this reason, a normalization of branch distance into the range [0,1] is crucial. Arcuri [3] analyses the flaws of the common used normalization function, proposing a different one. He compares them through an empirical study. The most used function, proposed by Baresel [57] was the following:

$$\omega_0(x) = 1 - \alpha^{-x}$$

where α is a constant that need to be set and x is the value to be normalized. A typical value is $\alpha = 1.001$ [57]. The different normalization function proposed by Arcuri [3] indeed is:

$$\omega_1(x) = \frac{x}{x + \beta}$$

where $\beta > 0$. Arcuri [3] compares them through an empirical study which shows that the new function ω_1 is more robust to rounding errors and has lower computational cost. The optimal choice of β is problem dependent but a common setting is $\beta = 1$.

2.3.5 Existing tools

Despite the large number of publications on Search-Based Software Testing, there remain few publicly available tools. To evaluate the applicability of different approaches, many tools were developed in the past, as TESTGEN [15], QUEST [8], ADTEST [19] or GADGET [44]. Unfortunately these tools are nowadays outdated, and, moreover, source code or binary is not available.

A recent and promising tool for automatically generating software test data, using search-based approaches is IGUANA [41]. Its fitness function is calculated by combining a branch distance measure with the approach level heuristic. Obviously, the goal of the search is to find the global minimum of the fitness function, i.e. zero. The approach level is a measure of how many control dependent nodes were not encountered in the path executed by the input vector. The branch distance indeed, expresses the closeness of an input

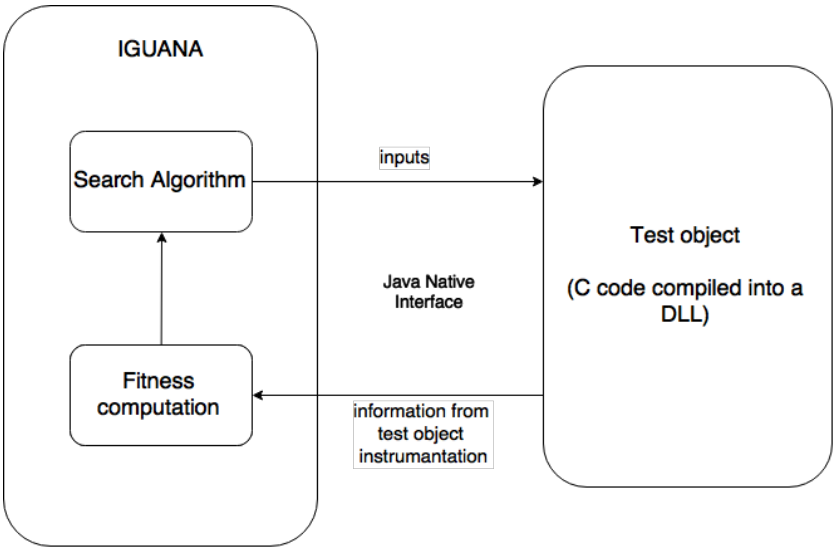


Figure 2.6: High level description of IGUANA

to satisfying the condition of the predicate at which point control flow for the test case went *wrong*.

In order to cover a particular branch in a unit under test, the goal is to construct an input vector which drives the execution of the program down the branch of interest. The test object is instrumented to return fitness information so search can use this heuristic to explore promising areas of the input domain. Figure 2.6 describes the entire behaviour of IGUANA.

In the instrumentation phase, each branching condition is replaced by a function call with two parameter, that replace the original one; obviously the instrumentation does not alter the code behaviour. The first parameter refers to the control flow graph node ID of the decision statement. The second represents the original boolean condition (the behaviour is correctly preserved). The instrumentation traces the execution of decision nodes and their outcome, i.e. branch distances, enabling the computation of approach level and branch distance.

Lakhotia et al. [36] recently propose an open source, search-based testing tool for C language called AUSTIN (AUgmented Search-based TestINg). It supports and uses three different search algorithms: the *random search*, the *hill climbing* in the form of the *Alternating Variable Method*, and a *hill climb-*

ing augmented with symbolic execution in order to handle constraints over pointer inputs to a function. The search is guided by the objective functions defined by Wagener et. al. [57].

Chapter 3

Search-based test data generation approaches

The final goal of automated structural testing is the derivation of test cases from the internal structure of the software under test, with the goal of maximizing the chosen coverage criterion. Coverage is the extent that a structure has been exercised as a percentage of items being covered. If coverage does not reach 100%, then more tests may be designed to test those items that were missed and therefore, increase coverage. There are some different coverage criteria, the main ones being:

- *statement coverage*: can only report whether a statements has been executed or not;
- *branch coverage*: aims to ensure that each one of the possible branch from each decision point is executed at least once;
- *condition coverage*: requires enough test cases such that each condition in a decision takes on all possible outcomes at least once;
- *path coverage*: test suite is executed in such a way that every path is executed at least once.

As seen in section 2.3, *structured-oriented* approaches uses the *divide et impera* technique to obtain full coverage and performs a separate search for each

intermediate objective required by the chosen coverage criterion, branches in *branch coverage*, code statements in *statement coverage* and so on.

This chapter introduces the current approach used in OCELOT, that has been proposed in Scalabrino's master thesis work [51]. This technique attempts to achieve the full branch coverage, trying to cover a set of linearly independent paths (see subsection 3.1.1). After this first step, the algorithm targets all previously uncovered branches with the aim to increase the coverage as possible. Then, the chapter describes two of most interesting novel approaches for automated test cases generation which has been recently proposed in latest literature. The first identified approach, proposed by Panichella et al. [47], proposes to address the branch coverage problem as a many-objective optimization problem, instead of a single-objective search performed multiple times, changing the target each time. A second approach, proposed by Harman et al. [27], focuses on control dependence graph analysis in order to maximise the amount of structural collateral coverage, reducing the number of test cases without loss of coverage. Both approaches have been implemented and evaluated in OCELOT with the aim to improve the effectiveness and the performance of the algorithm based on basis paths currently used in OCELOT.

3.1 McCabe's basis paths-based approach

As said before, a common problem of the *structure-oriented* approach is the selection of the intermediate objectives; the final aim is an overall level of coverage which is achieved through the coverage of this objectives. In the structural testing area, an intermediate target refers to any element, or set of elements, of a given program representation model, usually a *control flow graph*. A *control flow graph* (CFG) for a program P is a direct graph $G = (N, E, s, e)$ where N is a set of nodes, E is a set of edges, and s and e are entry and exit nodes, respectively. Each node $n \in N$ is a statement in the program, while each edge, $e = (n_i, n_j) \in E$, represents a transfer of control from node n_i to node n_j. Nodes corresponding to decision statements

(for example, an `if` or a `while` statement) are referred to as *branching nodes*. Outgoing edges from these nodes are referred to as *branches*. The condition determining whether a branch is taken is referred to as the *branch predicate*.

OCELOT defines and implements a novel approach [51] that uses as target both the McCabe's linearly independent paths and the single branches of the function under test. In order to understand that approach, the next paragraph reviews the concept of McCabe's basis path.

3.1.1 McCabe's linearly independent paths

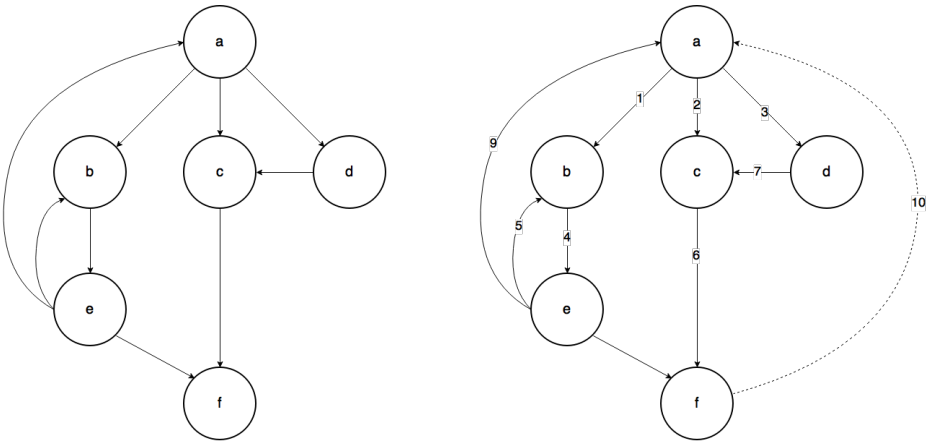
The cyclomatic complexity is a software metric developed by Thomas J. McCabe in 1976 [39] in order to measure and control the number of paths through a program. However, this approach raises a huge problem: the calculation of the total number of possible paths through a structured program is impractical. For this reason, the cyclomatic complexity measure is defined in terms of basis paths that, when taken in combination, will generate every possible path.
In order to understand the metric, the following preliminaries from the graph theory will be needed.

Definition 3.1. *In an undirected graph G, two vertices u and v are called connected if G contains a path from u to v; otherwise, they are called disconnected.* ■

Definition 3.2. *The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected component is*

$$V(G) = e - n + p$$

■



(a) (b)

Figure 3.1: Original control flow graph (3.1a) and modified control flow graph (3.1b) with an additional edge from the exit node f to the entry node a

Theorem 3.3. *In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits.*

The theorem 3.3 can be applied as follows. Given a program control flow graph, where each node corresponds to a block of code, it is assumed that each node can be reached by the entry node and each node can reach the exit node. With the connection of the exit node to the entry node, the graph becomes connected, i.e. there is a path joining any pair of distinct vertices, so the theorem 3.3 can be applied. The above concept can be seen in Figure 3.1; the figure on the left shows an example of control flow graph with an entry node *a* and an exit node *f*, while the figure on the right shows the same graph with the addition of the dotted edge (10) from the exit to the entry node. The maximum number of linearly independent circuits in the graph is $9 - 6 + 2 = 5$.

According to the theorem 3.3, it is possible to choose a basis set of circuits that correspond to the paths through the program. For example, the set

$$\mathbb{B} = \langle abef \rangle, \langle abeabef \rangle, \langle abebef \rangle, \langle acf \rangle, \langle adcf \rangle$$

refers to a set of basis paths for the program represented by the control flow

graph in Figure 3.1. A linear combination of $\mathbb{B}$ will also generate any path. For example the path $\langle abea(be)^3f \rangle$ can be generated as follow:

$$\langle abea(be)^3f \rangle = 2\langle abebef \rangle - \langle abef \rangle$$

The McCabe's *baseline* method is the approach used to determine a set of basis paths. It can be summarized in the following steps:

1. pick a *baseline* path that correspond to a normal execution; this *baseline* should have as many decisions as possible;
2. to get succeeding basis paths, retrace the *baseline* until a decision node is reached. *Flip* the decision (taking another alternative) and continue as much of the *baseline* as possible;
3. repeat this until all decisions have been flipped. Continue until $V(G)$ basis paths are reached;
4. if there aren't enough decisions in the first *baseline* path, find a second *baseline* and repeat the steps 2 and 3.

3.1.2 Basis paths coverage

The first step of the described approach aims to reach the maximum coverage through the execution of every linearly independent path. As seen in section 3.1.1, the execution of the basis paths guarantees the maximum *branch coverage*. At first, according to the baseline method seen above, all basis paths are generated. Then, a genetic algorithm is used in order to search for the appropriate input vector which allows the execution of the current target path. In order to lead the search toward a possible solution, the fitness function defined by Wagener et. al. [57] for the path-oriented approaches is used. The fitness function for path-oriented methods consists of two components, the *approximation level* and the *branch distance*.

Approximation level Let an *execution path* the path executed through a control flow graph. Let $e_n = \{e_1, ..., e_{n+1}\}$ the set of executed nodes where e_1

refers to the start node, while e_{n+1} refers to the end node. The *approximation level* is calculated by considering all branching nodes where the target path and the execution path diverge. Let $e_n^* = \{e_1^*, ..., e_n^*\}$ the set of nodes of the target path, the *approximation level* is calculated as follow:

$$al(e_n^*, e_n) = |e_n^* - e_n|$$

Branch distance Let D_{NN} the subset of the control flow graph nodes such that, $\forall NN \in D_{NN},\ e_e^* \ni E_\alpha = (NN, N_\alpha)$ and $e_e \ni E_\beta = (NN, N_\beta)$, with $E_\alpha \neq E_\beta$, i.e. the nodes whereas the paths e_n^* and e_n diverge. The branch distance relative to the execution path e_n and the target path e_n^* is calculated as follow:

$$bd(e_n^*, e_n) = \sum_{NN \in D_{NN}} bd(NN)$$

Fitness function Let T the target path and e_n the execution path, the overall fitness function is defined as follow:

$$f(e_n, T) = al(e_n, T) + normalized\ bd(e_n, T)$$

For this approach, the normalization function proposed by Arcuri [3] (see subsection 2.3.4 for more details) has been used.

Figure 3.2 shows an example of the fitness function calculation for the input vector $a = 5, b = 7, c = 4$. Grey nodes refers to the target path, while dashed nodes refers to the execution path. Since there are 2 nodes (nodes 1 and 5) which are in the target path and not in the execution path, the correspondent *approach level* value is 2. The *branch distance* is calculated by adding the branch distances relative to the nodes where the paths diverge; in this example, the paths diverges in nodes 1 and 5.

Table 3.2.b reports the calculated distances for the branching nodes. So, the resulting *branch distance* is $2.5 + 1.5 = 4$. In order to calculate the value of the objective function, the *branch distance* value is normalized and added

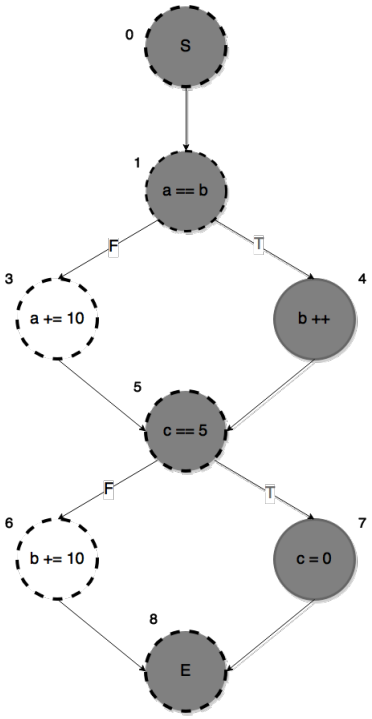
to the *approach level*. So,

$$f(e_n, T) = \frac{4}{4+1} + 2 = 2.8$$

3.1.3 Additional branch coverage

The approach based only on the basis paths presents some challenging problems. First of all, some paths could be hard to cover. Then, the problem of the *infeasible paths* must be addressed. Whenever the genetic algorithm fails to find the proper input needed to cover a specific target path, obviously the obtained level of coverage decreases.

In order to increase the overall branch coverage, once the search based on paths finishes, the algorithm identifies the uncovered branches and it



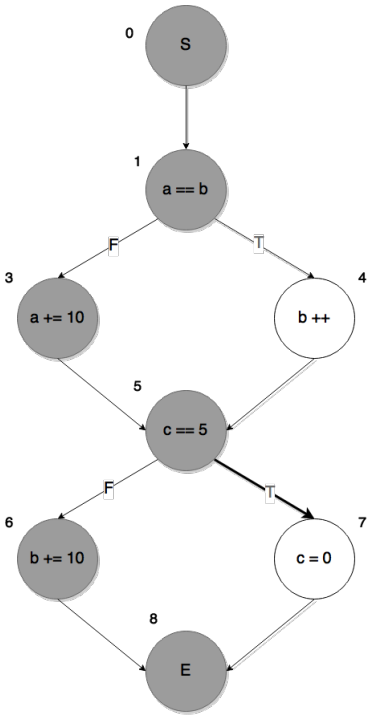
Node	Choice	T	F
1	0	2.5	0
5	0	1.5	0

Figure 3.2: Example of fitness function calculation for a target path. Grey nodes refers to the target path, while dashed nodes refers to the execution path for the input vector $a = 5, b = 7, c = 4$. The table on the right reports the choices done and the distances from `true` and `false` branches, for the function branching nodes.

attempts to cover them. The fitness function defined by Wegener el. al. [57] and used in the work of Harman et. al. [27] has been used to lead the genetic algorithm.

Approach level Let b_i the target branch and t_i the correspondent source node. The *approach level* is achieved by counting the number of unexecuted nodes on which t_i is control dependent (see definition 3.11). It is important to notice that t_i is necessarily a branching node.

Branch distance Let p_i the last predicate executed on a subpath from the start node to the target, i.e. the closer node to t_i. Here, it is possible to distinguish two different cases:



Node	Choice	T	F
1	0	2.5	0
5	0	1.5	0

Figure 3.3: Example of fitness function calculation for a target branch. Grey nodes refers to the execution flow for the input vector $a = 5, b = 7, c = 4$. The target branch (in bold) is the branch 5T. The table on the right reports the choices done and the distances from `true` and `false` branches, for the function branching nodes.

- $p_i \neq t_i$;
- $p_i == t_i$.

In the first case, if $p_i == t_i$, the resulting *branch distance* refers to the *branch distance* relative to the target b_i; otherwise, it refers to the branch distance relative to the node p_i.

Fitness function The fitness function for covering a target branch is the following,

$$f(b_i) = al(b_i) + normalized\ bd(b_i)$$

where the normalized function is the same function proposed by Arcuri [3] seen above.

Figure 3.3 shows an example of fitness function calculation for a target branch. Notice that the input vector is the following: $\langle a = 5, b = 7, c = 4 \rangle$. The grey nodes refers to the execution path, while the target branch is the `true` branch from the node 5. The nodes on which the target source node (the node 5) is control dependent are the node 0 and the node 1, which have been both executed. So, the resulting *approach level* is equal to 0. The *branch distance*, indeed, is directly obtained from the correspondent `true` distance of the node 5, so it is equal to 2.5. The resulting objective function is calculated by summing the *approach level* value and the normalized *branch distance*:

$$f(e_n, T) = \frac{1.5}{1.5 + 1} + 0 = 0.6$$

3.2 MOSA: a Many-Objective Sorting Algorithm

Branch coverage problem can be described as: find a set of test cases which maximize the number of covered branches in code under test. With this approach, a typical solution can be summarized in the following steps:

- enumerate all targets (branches);

- perform a single-objective search multiple times, for each target, until all targets are covered or the total search budget (time or maximum number of iteration) is consumed;
- combine all generated test case in a single test suite.

However, this strategy presents several issues [18], such as the different effort needed to cover different targets, or a target infeasibility. To overcome these limitations, recently Fraser and Arcuri [18] have proposed an approach, called *whole suite* which combine the fitness function of every single branch into an aggregate test suite-level fitness, considering all testing goals simultaneously. In this approach, they refers to a candidate solution as a test suite instead of a single test case.

Starting from this idea, Panichella et. al [47] propose to consider branch coverage problem directly as a many-objective optimization problem, instead of aggregating multiple objectives into a single values, like *whole suite* approach. In this reformulation, a test case become a candidate solution, while *fitness* is evaluated according to all branches at the same time. They introduces a novel many-objective algorithm, named MOSA (Many-Objective Sorting Algorithm) which applies the above reformulation. This approach produces better results compared than *whole suite*, i.e. a higher coverage or, with the same achieved coverage, a faster convergence [47].

3.2.1 Problem reformulation

As said before, Panichella et al. reformulate the branch coverage criterion as a many-objective optimization problem, where the objective functions to be minimized are the branch distances of all the branches in the test object. In more formal terms, the reformulation proposed is the following:

Problem 3.4. *Let $B = \{b_1, b_2, ..., b_m\}$ be the set of branches of the test object. Find a set of non-dominated test cases $T = \{t_1, t_2, ..., t_n\}$ that minimize the*

fitness functions $\forall b_i \in B$*, i.e. minimizing the following* m *objectives:*

$$\begin{cases} min\ f_1 = al(b_1, t) + d(b_1, t) \\ min\ f_2 = al(b_2, t) + d(b_2, t) \\ \vdots \\ min\ f_m = al(b_m, t) + d(b_m, t) \end{cases}$$

where $d(b_i, t)$ *refers to the normalized branch distance of test case* t *for branch* b_i*, while* $al(b_i, t)$ *refers to the corresponding approach level.*

It is important to notice that, in this technique, the *approach level* and the *branch distance* are calculated in the same way as seen in subsection 3.1.3.

In contrast with *whole suite*, this reformulation provides a candidate solution as a test case instead of a test suite and its fitness is evaluated calculating *branch distance* and *approach level* for each branch in test object. So, in this approach, a fitness vector $\langle f_1, ..., f_m \rangle$ of m values represents the overall evaluation of a solution.

In order to understand the evaluation mechanism in many-objective optimization, it is necessary to introduce the definitions of *Pareto dominance* and *Pareto optimality* [10].

Definition 3.5. *A test case* x *dominates another test case* y *(also written* $x \prec y$*) if and only if the values of the objective functions satisfy the following conditions:*

$$\forall i \in \{1, ..., m\},\ f_i(x) \leq f_i(y)$$

and

$$\exists j \in \{1, ..., m\}\ s.t.\ f_j(x) < f_j(y)$$

■

Definition 3.6. *A test case* x^* *is Pareto optimal if and only if it is not dominated by any other test case in the space of all possible test cases.* ■

To put it in a nutshell, definition 3.5 indicates that x dominates y if and only if x is better on one or more objectives and it is not worse for the

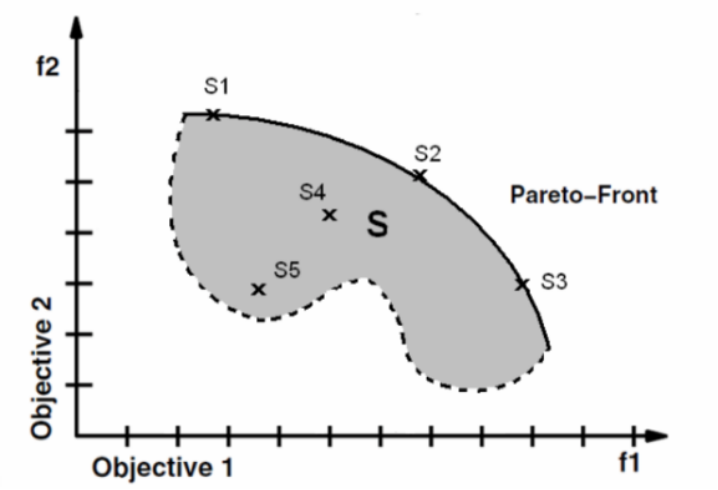


Figure 3.4: Pareto optimality and Pareto fronts

remaining; definition 3.6 indeed, refers to a set of optimal test cases which are non-dominated by any other possible test case.

Figure 3.4 shows an example of Pareto optimal front and dominance relations in a two-objective space for a problem with two different criteria. In the figure, points $S1$, $S2$ and $S3$ lie on the pareto front, while $S4$ and $S5$ are dominated.

Usually, in multi-objective problems, a solution leads to a set of Pareto-optimal solutions which corresponds to trade-offs in the feasible region. However, considerations among trade-offs in the context of structural testing are totally useless since the final goal is to find only the solutions (test cases in this context) which contribute to maximize the total coverage. In other words, we are looking for solutions having at least one objective function equal to zero, i.e. $f_i(t) = 0$. Conceptually, these test cases intersect one or more of the m Cartesian axes of the vector space; they represent a sub-set of the Pareto-optimal solution and they probably should be included in the final test suite.

3.2.2 The algorithm

Previous works on many-objective optimization demonstrated that many multi-objective algorithms, such as NGSA-II (Non-dominated Sorting Genetic Algorithm II) [11] or SPEA2 (Strength Pareto Evolutionary Algorithm) [64] are not effective in solving optimization problems with a significant number of objectives, typically just more than three [47].

To overcome this limitation, new algorithms have been proposed in last years. For example, Laumanns et al. [37] proposed the usage of an ϵ-dominance relation instead of the classical one; Zitzler and Künzli [65] proposed the usage of the hypervolume indicator instead of the Pareto dominance when selecting the best solution to form the next generation; Yuan et al. [61] indeed, proposed θ-NGSA-III, an improved version of the classical NGSA-II, where the non-dominated sorting scheme is based on the concept of θ-dominance to ensure convergence and diversity.

However, all algorithms mentioned presents two main limitations: they have been used mostly for problems with less than 15 objectives (typically in structural testing the overall number of objectives could be much bigger) and they have been designed to produce a set of optimal trade-offs between different optimization goals. As explained in section 3.2.1, in structural testing we are interested in finding only the test cases that intersect with any of the m Cartesian axes in the vector space (a sub-set of the optimal solutions), so considering trade-offs is useless, except for maintaining diversity, and not all Pareto optimal test cases have a practical utility.

Preference criterion

Previous research [38] revealed that many-objective problems present a huge challenge: increasing the number of objectives, the proportion of non dominated solutions increases exponentially. In this scenario, assigning a preference among individuals for selection purpose become almost impossible and the overall search process become equivalent to a random one [38].

So, with the aim to determine a domain specific order of preference over non-dominated test cases, Panichella et. al. [47] proposes a novel *preference*

criterion:

Definition 3.7. *Given a branch b_i, a test case x is preferred over another test case y (also written $x \prec_{b_i} y$) if and only if the objective function for b_i, satisfies the following condition:*

$$f_i(x) < f_i(y)$$

where $f_i(x)$ refers to the objective score of test case x for branch b_i. ■

They refer to the *best test case* for the one preferred over all the others for a given branch, i.e. $x_{best} \prec_{b_I} y, \forall y \in T$. When there are multiple test cases with the same minimum fitness value for a given branch, Panichella et. al. [47] use the test case length as a secondary preference criterion. Since OCELOT has been designed for C language, each test case trivially has no length, so this criterion is not applicable. Since the authors refer to Java language, they define this secondary criterion because, in their case, a test case is composed by a set of lines of code (usually object allocations and method calls instructions) testing a given class; in this scenario it is trivial to choose the shorter between two test cases with the same fitness value. The set of best test cases across all uncovered branches define a subset of Pareto front that has higher priority and higher probability to be selected for the next generation than the other non-dominated test cases.

Algorithm 4 shows pseudo-code for the *preference criterion* above. As said before, this criterion provides an efficient way to rank test cases in a set of non-dominates ones, increasing the selection pressure giving higher priority to best test cases. Lines from 2 to 6, determine the test case with the lowest objective score, calculated summing *branch distance* and *approach level*, for each b_i, i.e. for each uncovered branch. The resulting test case represent the closest ones to cover b_i. The algorithm assigns rank 0 to these candidate solutions, inserting them into the first non-dominated front $\mathbb{F}_0$. This technique, named *elitism*, allows the solutions that have been inserted in this first front, to be selected with higher probability for the next generation The remaining test cases are ranked according to the traditional non-dominated sorting algorithm used by NGSA-II [11].

Algorithm 4: Preference-Sorting Algorithm

```
Input: T: a set of candidate test cases
Result: 𝔽: non-dominated ranking assignment
begin
1   | 𝔽_0                                  /* first non-dominated front */
2   | for b_i ∈ B and b_i is uncovered do
3   |   | /* for each uncovered branch, select the best test
    |   | case according to the preference criterion          */
4   |   | t_best ← test case with minimum objective score for b_i
5   |   | 𝔽_0 = 𝔽_0 ⋃ {t_best}
    | end
6   | T ← T − {t_best}
7   | if T is not empty then
8   |   | 𝔾 ← T
9   |   | FAST-NONDOMINATED-SORT(T, {b ∈ B | b is uncovered})
10  |   | d ← 0                                /* first front in 𝔾 */
11  |   | for all non-dominated fronts in 𝔾 do
12  |   |   | 𝔽_{d+1} ← 𝔾_d
    |   | end
    | end
end
```

The routine FAST-NONDOMINATED-SORT, whose pseudo-code is provided in Algorithm 5, ranks the test cases not assigned to the first front. It is important to notice that, differently from NGSA-II implementation, in MOSA this step start with a rank equal to 1, because the rank 0 is designed for those solutions selected by the novel preference criterion.

FAST-NONDOMINATED-SORT routine requires $O(MN^2)$ computation time. First, for each solution it calculates two entities: n_p, i.e. the number of solutions which dominate the solution p, and S_p, a set of solutions that the solution p dominates. All solutions in the first non-dominated front have their domination count as zero. For each p with $n_p = 0$, the algorithm visits each member $q \in S_p$ and reduces its domination count by one. In doing so, the algorithm puts any member q whose the domination count become zero, in the list Q; these solutions belongs to the second non dominated front. This process continues until all fronts are identified.

Algorithm 5: Fast-Non-Dominated-Sort Algorithm

```
Data: P: a set of candidate test cases
foreach p ∈ P do
    S_p = 1
    n_p = 0
    foreach q ∈ P do
        if p ≺ q then                                   /* if p dominated q */
            S_p = S_p ⋃ {q}          /* add q to the set of solutions dominated by p */
        else if p ≺ q then
            n_p = n_p + 1                /* increment the domination counter of p */
    end
    if n_p = 0 then                                     /* p belongs to the first front */
        p_rank = 1
        F_1 = F_1 ⋃ {p}
    end
end
i = 1                                                   /* initialize the front counter */
while F_i ≠ ∅ do
    Q = ∅                              /* used to store the members of the next front */
    foreach p ∈ F_i do
        foreach q ∈ S_p do
            n_q = n_q − 1
            if n_q = 0 then                             /* q belongs to the next front */
                q_rank = i + 1
                Q = Q ⋃ {q}
        end
    end
    i = i + 1
    F_1 = Q
end
```

It is also important to notice that the FAST-NONDOMINATED-SORT routine in MOSA algorithm assigns the ranks by considering only the non-dominance relation for the uncovered branch, focusing the search toward the interesting sub-region of the search space.

A graphical interpretation In order to better understand the proposed *preference criterion*, a graphical comparisons between MOSA ranking algorithm and traditional ranks assignment may be necessary. Let us consider the example program shown in Listing 3.1.

Let us assume that the uncovered targets are the `true` branch from node 2 to node 3 and the `true` branch from node 4 to node 5, whose predicates are `(a==b)` and `(b==c)` respectively. The corresponding many-objective op-

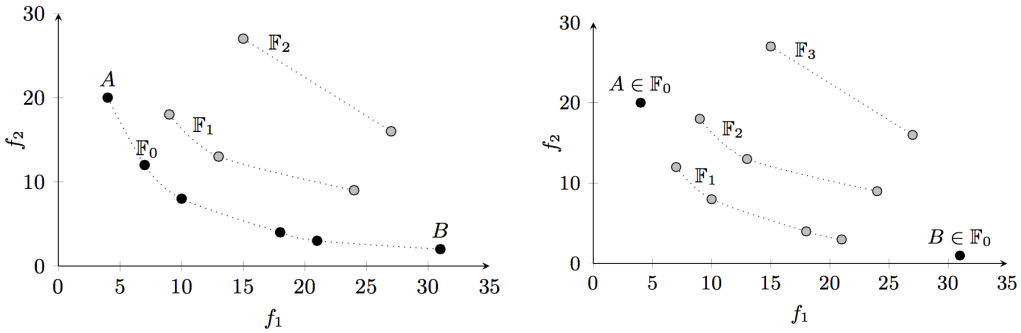
```
1 int example (int a, int b, int c) {
2 if (a == b)
3     return 1;
4 if (b == c)
5     return -1;
6 return 0;
7 }
```

Listing 3.1: Example program

timization problem has two residual goals, namely:

$$\begin{cases} f_1 = al(b_1) + d(b_1) = abs(a - b) \\ f_2 = al(b_2) + d(b_2) = abs(b - c) \end{cases}$$

Since there are two remaining goals, any solution produced corresponds to a single point in a two-dimensional search space which reports the objective functions f_1 and f_2 on the horizontal and vertical axis, respectively. Figure 3.5a and 3.5b shows the search space above. First, let us consider the scenario in Figure 3.5a: no test case represents a solution which can cover the two remaining branches, trivially because any solution intersects a Cartesian axis, i.e. f_1 and f_2 are greater than 0 for all test cases. Using the traditional non-dominance relation, all solutions, corresponding to the black points in figure



(a) Ranking based on the traditional non-dominance relation

(b) Ranking based on the MOSA preference criterion

Figure 3.5: Comparison between the non-dominated ranks assignment obtained by the traditional non-dominated sorting algorithm and the the ranking algorithm based on preference criterion proposed in MOSA algorithm

3.5a are non-dominated and form the first non-dominated front $\mathbb{F}_0$. However, even if some test cases are closest to the axes (for example, the test case A is closer to cover the branch b_2 than other solutions in $\mathbb{F}_0$), all solutions in first non-dominated front have the same probability to be selected to form the next generation. Since there is no preference among solutions in $\mathbb{F}_0$, it might happen than test case A, which is a better solution for branch b_2, is not kept for the next generation, while other test cases less useful in $\mathbb{F}_0$ are preserved. Several studies [38] show that this scenario is common in problems where the number of non-dominated solutions increases with the number of objectives.

The *preference criterion* proposed by Panichella et. al. [47] addresses this problem distinguishing between solutions in a set of non-dominated ones. Figure 3.5b shows the same example scenario, ranking the test cases with this novel criterion: it gives higher priority to test case A and B, guaranteeing their survival in the next generation. In fact in this case, unlike traditional non-dominance relation, the first non-dominated front $\mathbb{F}_0$ contains only test cases A and B that are the closest to the Cartesian axes, while other test cases are assigned to successive fronts.

MOSA: a New Many Objective Sorting Algorithm

Algorithm 6 shows the high level description of the MOSA algorithm. It starts with an initial set of randomly generated test cases that forms the *initial population* (see line 3 of Algorithm 6). In order to evolve this initial population towards better test cases, the algorithm needs to produce a series of *generations*. Doing this, MOSA first creates *offsprings*, i.e. new test cases, combining two selected ones, called *parents*, of current generation, using the *crossover* operator and randomly modifying test cases using a *mutation* operator (line 6 of Algorithm 6). Then, a new population is generated performing a *selection* from parents and offspring, considering both the non-dominance relation and the proposed *preference criterion*, as previously discussed (line 8 of Algorithm 6). As seen before, the PREFERENCE-SORTING function, which can be seen in Algorithm 4, assigns rank 0 (inserting them into the first non-dominated front $\mathbb{F}_0$) to test cases with the lowest objective score for

Algorithm 6: MOSA

Input: $B = \{b_1, ..., b_m\}$: the set of branches of a program
Population size M
Result: A test suite T

```
1  begin
2      t ← 0                                        /* current generation */
3      P_t ← RANDOM-POPULATION(M)
4      archive ← UPDATE-ARCHIVE(P_t)
5      while not(search budget consumed) do
6          Q_t ← GENERATE-OFFSPRING(P_t)
7          R_t ← P_t ⋃ Q_t
8          𝔽 ← PREFERENCE-SORTING(R_t)
9          P_{t+1} ← ∅
10         d ← 0
11         while |P_{t+1}| + |𝔽_d| ≤ M do
12             CROWDING-DISTANCE-ASSIGNMENT(𝔽_d)
13             P_{t+1} ← P_{t+1} ⋃ 𝔽_d
14             d ← d + 1
           end
15         Sort(𝔽_d)          /* according to the crowding distance */
16         P_{t+1} ← P_{t+1} ⋃ 𝔽_d[1 : (M − |P_{t+1}|)]
17         archive ← UPDATE-ARCHIVE(archive ⋃ P_{t+1})
18         t ← t + 1
       end
19     T ← archive
   end
```

each uncovered branch b_i, and then it ranks the remaining test cases assigning them to successive fronts.

Once a rank is assigned to all candidates, the *crowding distance* [11] is used. The crowding distance is a measure of how close an individual is to its neighbours. Large average crowding distance will result in better diversity in the population, which represent a key factor to avoid premature convergence toward suboptimal solutions [31]. So, this measure is used in order to make a decision about test case selection: a higher distance from the rest of the population gives higher probability to be selected. The algorithm 7 outlines the crowding-distance computation procedure of all solutions in a

Algorithm 7: Crowding distance assignment

```
crowding-distance-assignment(I);
l = |I| ;                          /* number of solutions in I */
foreach i do
    set I[i]_distance = 0 ;              /* initialize distance */
end
foreach objective m do
    I = sort(I, m) ;    /* sorting using each objective value */
    I[1]_distance = I[l]_distance = ∞ ;   /* boundaries always selected
    */
    for i = 2 to (l − 1) do              /* for all other points */
        I[i]_distance = I[i]_distance + (I[i + 1].m − I[i − 1].m)/(f_m^max − f_m^min);
    end
end
```

non-dominated set I.

To get an estimate of the density of solutions surrounding a particular ones, it needs a quantity, $i_{distance}$ which refers to the average distance of two points on either side of these points along each of the objectives. Namely, this quantity estimates the perimeters of the cuboid formed using the nearest neighbours as vertices. Figure 3.6 shows the concept above for the ith solution in its front; the crowding distance is the average side length of the cuboid shown with a dashed box.

The crowding-distance first requires sorting the population according to each function value in ascending order of magnitude. Thereafter, for each objective function, the boundary solutions are assigned as infinite distance value. The algorithm assigns to all intermediate solutions a distance value equal to the absolute normalized difference in the function values of two adjacent solutions. This calculation continues with other objective functions. The overall crowding-distance value is calculated as the sum of individual distance values corresponding to each objective. The final crowded-comparison operator works in the following way: between two solutions with differing non-domination ranks, it prefer the solution with lower one; otherwise, if both solutions belong to the same front, it prefers the solution located in a lesser crowded region.

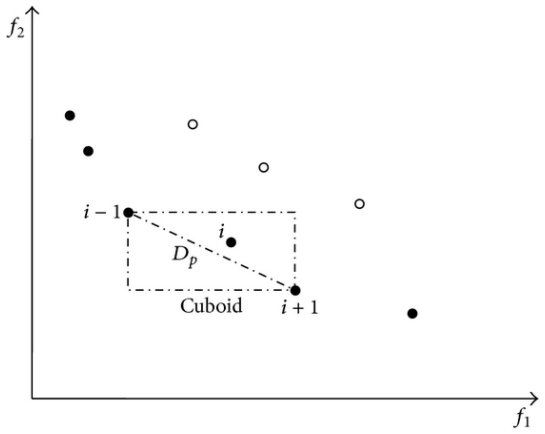


Figure 3.6: Crowding-distance calculation. Filled points are solutions of the same non-dominated front

So, a combination of non-dominated sorting with the proposed *preference criterion* and *crowding distance sorting* allows to find the individuals to be selected for the next generation. Algorithm 6 performs this process in the loop at line 11 and in the following lines 15 and 16, adding as many test cases as possible to the next generation, until reaching the population size. It first selects test cases from $\mathbb{F}_0$ and so on, not exceeding the population size. Then (lines 15-16), when the number of test cases selected is lower than M, it fills the next generation selecting the remaining test cases from the current front, according to a descending crowing distance order. Figure 3.7

Algorithm 8: Update Archive

```
Input: A set of candidate test cases T
Result: An archive A
A ← ∅
for b_i ∈ uncovered branches do
    t_best ← ∅
    length_best ← ∞
    for t_j ∈ T do
        score ← objective score of t_j for branch b_j
        length ← number of statements in t_j
        if score == 0 and length ≤ lenght_best then
            t_best ← {t_j}
            lenght_best ← length
    end
    if t_best ≠ ∅ then
        A ← A ⋃ t_best
end
```

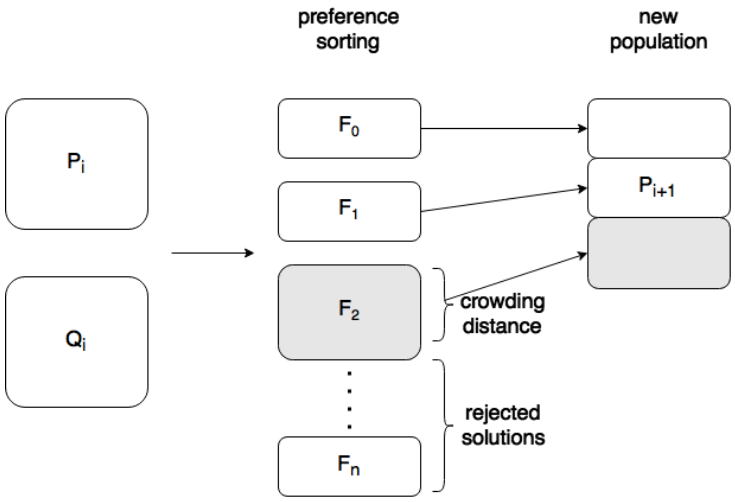


Figure 3.7: Selection procedure for next generation

shows the entire process of test cases selection.

The second peculiarity of MOSA algorithm over other many-objective algorithms, is the usage of a second population, called *archive*, that is used to track the best test cases which cover a branch in test object. In order to do this, at the end of each generation, the function UPDATE-ARCHIVE (Algorithm 8) updates the set of stored test cases which are candidates to form the final test suite. Since the MOSA algorithm was originally developed in order to test Java code and it was implemented in a prototype tool by extending the EvoSuite [18] test data generation framework, it also considers the length of test cases when updating the *archive*: for each covered branch b_i it stores the shortest test case covering b_i in the archive. Obviously, the correspondent OCELOT implementation of the UPDATE-ARCHIVE functions works a bit differently (see subsection 4.2.6 for more details).

3.3 CDG-based approach

In the last years, many search based approaches fro test data generation addressed the problem of maximizing coverage on test objects. However, structural testing presents also another challenging problem, the oracle cost

reduction, i.e. the oracle definition for each test case in a test suite. While, for a system with precise models, checkable specifications or contract driven development, the oracle definition could be automatized, in many real cases this remain an illusion and usually system behaviour must be checked by human. This process forms a significant cost, called the oracle cost. The simplest way to reduce it consists of reducing the number of test cases generated. Harman et. al. [27] introduce a new formulation of search based structural test data generation problem, in which the goal is to maximize coverage, while minimizing the number of test cases; they introduce three algorithms for addressing this re-formulation. The first algorithm, called *Memory-based approach*, is simply a codification of commons sense and it is used merely as a baseline against which to compare the other two algorithms. It simply records all branches hit by a test case that covers a particular branch of interest. The second algorithm, named *Greedy-Approach*, for each branch in the target program, first generates a set of test cases using a search algorithm. In a second stage, it applies a greedy-based test suite minimisation to the pool of test cases. This sections focuses on the third of these algorithms, a control dependence graph based test data reduction approach.

3.3.1 Background concepts

This subsection presents a brief recall about some static analysis techniques used in the presented approach. Below, some important definitions are enunciated.

Definition 3.8. *A node i dominates a node j if and only if every path from the entry node to the node j passes through node i.* ■

Definition 3.9. *A node j post-dominates a node i if and only if every path from the node i to the exit node traverses the node j.* ■

Note that this definition of post-dominance does not include the initial node on the path. In particular, a node never post-dominates itself.

Definition 3.10. *A node k post-dominates a branch $e = (i, j)$ if and only if every path from the node i to the exit node through e contains the node k.* ■

Definition 3.11. *A node j is control dependent on a node i if and only if the node i dominates the node j and the node j post-dominates one and only one of the branches of the node i.* ■

A Control Dependent Graph (CDG) represents a program as a graph in which the nodes are statements and predicate expressions and the edges incident to the node represent both the data values on which the node's operations depend and the control condition on which the execution of the operations depends [16]. So a CDG is an intermediate program representation that captures control dependences, explicating both the data and control dependence for each operation in a program derived from the usual control flow graph. Since dependencies in the CDG connect computationally related parts of the program, many traditional optimizations operate more efficiently on it. It is important to notice that sibling nodes in a CDG, that belong to one parent node, connected through the same edge, must either have a dominance or post-dominance relationship between each other. This means that, if one of these nodes is executed, then the other sibling nodes must also be executed. Figure 3.8 shows a version of a triangle classification problem with its relative CFG, while Figure 3.9 shows the relative control dependence graph. For example, in the triangle program, node 23 in control dependent on node 21, which in turn is control dependent on node 14. Node 14 itself has no control dependencies, other than the entry node which causes the function to be executed.

As said previously, the most common fitness function for covering a target branch requires two components: an approach level and a branch distance. From definition 3.11 it is possible to describe the approach level as the number of unexecuted nodes on which the target node is control dependent. Branch distance indeed refers to the *closeness* in satisfying the last conditional expression executed on the path from the entry node to the target.

```
1  int triangle(int a, int b, int c)
2  {
3      int type;
4
5      if (a > b) {
6          int t = a; a = b; b = t;     }
7
8      if (a > c) {
9          int t = a; a = c; c = t;     }
10
11     if (b > c) {
12         int t = b; b = c; c = t;     }
13
14     if (a + b <= c)
15     {
16         type = NOT_A_TRIANGLE;
17     }
18     else
19     {
20         type = SCALENE;
21         if (a == b && b == c)
22         {
23             type = EQUILATERAL;
24         }
25         else if (a == b || b == c)
26         {
27             type = ISOSCELES;
28         }
29     }
30     return type;
31 }
```

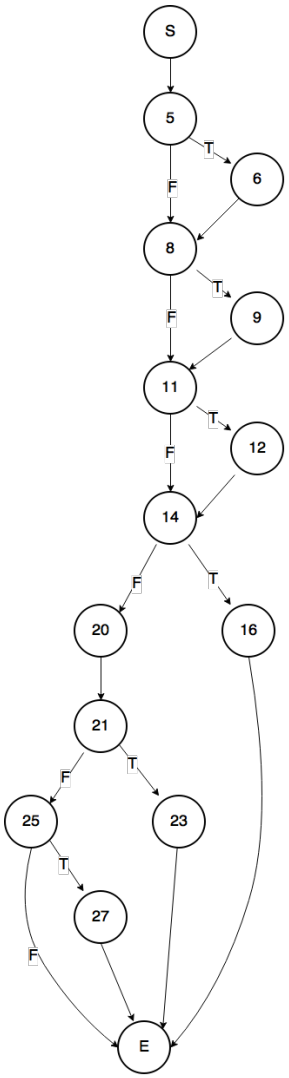


Figure 3.8: A triangle classification program and its corresponding CFG

3.3.2 CDG-Based Test Data Reduction

Any standard SBST approach performs a distinct search process for each uncovered target. Clearly, this solution is not optimal from the point of view of reducing the number of test cases required to reach the highest coverage for the given test object. Harman et. al. [27] formalize the trivial idea to record also all branches hit by a test case that covers the target branch for which the search process has been performed.

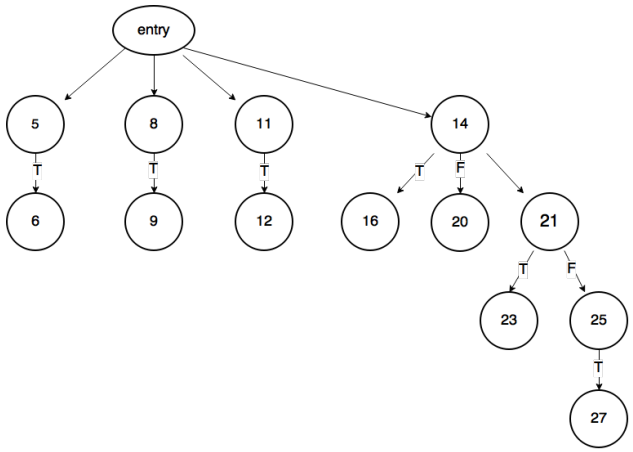


Figure 3.9: Control dependence graph for the triangle classification program in Figure 3.8

Algorithm 9 shows the pseudo-code which formalizes this concept. Simply, $P(x)$ denotes all branches covered by the execution of the test object P with the input x, which will correspond to a single test case. Whenever the program runs with the input x, specifically generated for the target t, the algorithm also removes from the set of remaining target, U, all branches *collaterally* covered by the execution of P with input x; obviously those branches will not be targeted again, allowing to reduce the 1:1 ratio between targets and resulting test cases.

The CDG-based approach, in order to reduce the size of the resulting

Algorithm 9: Memory-based approach

Input: P: target program; B: set of all branches in P
Output: C: set of test cases
$U \leftarrow B$
$C \leftarrow \emptyset$
while $U \neq \emptyset$ **do**
  select a $t \in U$
  search for x s.t. $t \in P(x)$
  if x *is found* **then**
    $U \leftarrow U - P(x)$
    $C \leftarrow C \bigcup \{x\}$
  else
    $U \leftarrow U - \{t\}$
  end
end
return C

Algorithm 10: CDG-based approach

Input: P: target program; B: set of all branches in P
Output: C: set of test cases

```
1  U ← B
2  C ← ∅
3  while U ≠ ∅ do
4  |  select a t ∈ U
5  |  search for x s.t. t ∈ P(x)∧ maximizes |P(x) ⋂ U|
6  |  if x is found then
7  |  |  U ← U − P(x)
8  |  |  C ← C ⋃ {x}
   |  else
9  |  |  U ← U − {t}
   |  end
   end
10 return C
```

test suite, uses the memory-based idea, but it also considers the amount of *extra* structural coverage that a solution can achieve. This approach is formalized in Algorithm 10. The main difference between this approach and the memory-based one is in line 5: the CDG-based approach focuses not only in achieving coverage for the target branch, but it continues its search, also after that t has been covered, in order to maximise the amount of possible collateral coverage. Obviously, the overall algorithm is based on the control dependence graph [16] representation of test object, in order to calculate the correct amount of collateral coverage which can be achieved by an input x.

Harman et. al [27] define an easy way to calculate the number of potentially coverable edges when targeting a specific edge e, first for program without loop and then, relaxing the definition, also for programs which contain loop control structures.

Coverable Branches First of all, it is possible to observe the Figure 3.10 in order to better understand how the Control Dependence Graph is used to calculate coverable branches; it shows the CFG and the CDG of the same program. In that figure, the edges which have been already covered are

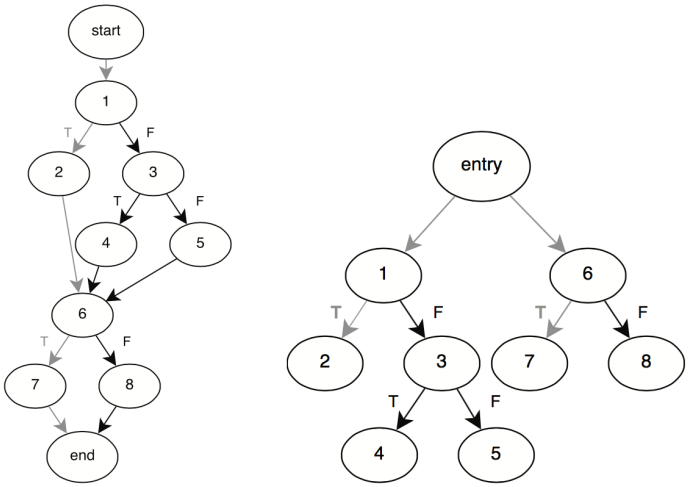


Figure 3.10: CFG (left) and CDG (right) that are used to show how coverable branches are calculated

coloured in grey, while edges to cover yet are coloured in black. Suppose that the next target branch is 1F, i.e. the false branch from node 1 to node 3. The set of target's post-dominating nodes contains node 6 and the exit node; the edges 1F, 3T and 3F are control dependent on node 1 and the edge 6F is control dependent on node 6. It is important to notice that those branches can be covered *serendipitously* [57] while targeting the branch 1F. Since, being mutually exclusive, with a single test case, only one branch between 3T and 3F can be covered, the number of additional coverable branches when attempting the branch 1F is 2 (1F and either 3T or 3F).

Harman et. al [27] formalizes a recursive calculation for number of potentially coverable edges. Suppose that n refers to a node in a CDG program representation and M indicates the set of edges that are already covered. Then, for branching nodes, E represents the `true` and `false` branches, i.e., $E = \{e_t, e_f\}$; for $e \in E$, let N_e be the set of nodes dominated by the edge e for the node n. In a similar way, let $L_e(n)$ be the number of potentially coverable edges when targeting the edge e. This value can be recursively

calculated as follow:

$$L_e(n) = \begin{cases} 0 & \text{if } n \text{ isn't a branching node} \\ \sum_{n_i \in N_e} max(L_{e_t}(n_i), L_{e_f}(n_i)) & \text{if } e \in M \\ 1 + \sum_{n_i \in N_e} max(L_{e_t}(n_i), L_{e_f}(n_i)) & \text{if } e \notin M \end{cases}$$

Fitness Function for Collateral Coverage Once the number of potentially coverable branches, a.k.a. $L_e(n)$, is calculated, it is possible to formally express the collateral coverage calculation. Harman et. al [27] define the collateral coverage as follows:

$$C(x, n, e) = \frac{|P(x) - M|}{L_e(n)}$$

where x is the candidate input, e is the target branch and n is the source node of the edge e. The numerator refers to the number of edges which are newly covered by the execution with the input x. It is important to notice that the quantity $|P(x) - M|$ is always less or equal than $L_e(n)$ for programs without loops. This fact is true if the edges in CDG are targeted in top-down order: this implies that any newly covered edges by the input x is also control dependent on the target e.

Using the above definition of collateral coverage, Harman et. al [27] formalize a new definition of the standard fitness function as follow:

$$f(x, n, e) = al(x) + bd(x) + (1 - C(x, n, e))$$

where $f(x, n, e)$ represents the overall fitness function, $al(x)$ refers to the approach level and $bd(x)$ refers to the normalized branch distance. Despite this normalization seems to be less efficient [3], they normalize as follow:

$$normalize(d) = 1 - 1.001^{-d}$$

This fitness function is composed of two parts, the first which includes approach level and branch distance, and the second which refers to $1-C(x, n, e)$.

Obviously the objective is to minimize it; the function reaches the overall value zero when the input x covers e and achieves the maximum number of potentially coverable branches. It is important to notice that only the first part provides the guidance towards the target coverage, while the second part consists only in a measurement of the collateral coverage that has been achieved. So, the primary fitness function is alone applied until the original target is covered. Then, once the search finds a solution which achieve the target, the secondary fitness is applied in order to determine the best candidate according to the amount of collateral coverage achieved.

The definition of $L_e(n)$ relies on the assumption that a single execution can cover only the `true` or the `false` branch of a predicate node. Obviously this assumption is not valid for loop predicates or for predicates contained within a loop. In this case, a single execution can naturally execute both the `true` and `false` branch. This means that, in the above situation, since the numerator value $|P(x)-M|$ could be higher than $L_e(n)$, a resulting collateral coverage higher than 1 is possible. To avoid this problem, Harman et. al [27] relax the previous definition of $C(x, n, e)$ as follow:

$$C(x,n,e) = \begin{cases} 1 & \text{if } |P(x) - M| > L_e(n) \\ \frac{|P(x)-M|}{L_e(n)} & \text{otherwise} \end{cases}$$

Chapter 4

Tool architecture and implementation

The tools listed in section 2.3.5 present several problems which do not allow their usage in this work. For most of them, the source codes are not public available or only binaries are provided. Moreover, original developers are usually reluctant to grant the full usage of their software. The few open source codes found after a detailed research, although potentially very promising, are unusable for the purpose of this work, for some reasons. For example, some of them have been developed in an uncommon languages, e.g. Austin [36], which has been developed in OCaml.

For these reasons, in order to aid future experimental studies, a new tool has been implemented completely from scratch. The development process started after a detailed analysis of IGUANA's [41] basis concepts. The tool is called OCELOT and its name originates from the ocelot, a feline which is a natural predator of iguana. It implements several approaches, including a technique based on McCabe paths coverage proposed by Scalabrino [51] in his work of master thesis, that is able to generate a test suite with the aim of obtaining the highest coverage on functions developed in C language. This approach represents the standard one used in OCELOT. As said, the aim of this work is to introduce and implement in OCELOT the latest methodologies raised by the most recent literature, in order to improve its effectiveness and

efficiency. OCELOT is developed in Java language. The high availability of open-source components needed to speed up the development processes addressed this choice.

This chapter describes the OCELOT architecture, its main components, and its execution process.

4.1 Open Source Components

The first step of OCELOT development process was an detailed investigation of all reusable components (library or framework) free available, useful to speed up the implementation. The following principal components have been identified: CDT for the analysis of C source codes, jMetal [14] to facilitating meta-heuristics development and jGraph which is a graph drawing open source component written in Java. Other minor libraries, such as Apache Common Maths, a library of mathematics and statistics components, have been used.

4.1.1 CDT

The C/C++ Development Toolkit (CDT) is a collection of Eclipse-based features that provides the capability to create, edit, navigate, build and debug projects that use C and/or C++ as a programming language. CDT offers many classes which can facilitate the static analysis of C source codes. In particular, the following CDT functionalities and components have been used:

- a parser able to extract the *Abstract Syntax Tree* from a specified piece of code. CDT also offers a mechanism based on the visitor pattern for the tree traversal of the AST;
- the association of entity with identifiers (binding);
- visitors for the AST translation into C source code.

First, the class `GCCLanguage` has been used to extract the *abstract syntax tree* for the specific function under test, starting from a C file and from all

Method	**Description**
visit(IASTTranslationUnit)	Visits the root node in the physical AST. The translation unit represents a compilable unit of source
visit(IASTDeclaration)	Visits a declaration, such as a function or a variable definition
visit(IASTStatement)	`IASTStatement` is the root interface for statements. This method visits all kind of expressions, such as assignments or control structures
visit(IASTExpression)	Visits a general expressions, that is a piece of an instruction like a binary or unary expression (a condition) or a cast expression

Table 4.1: Principal method of `ASTVisitor` class

folders which contain the files to include (headers). The *abstract syntax tree* representation is crucial for a easily analysis of the program. In particular, OCELOT uses the *abstract syntax tree* in order to perform two crucial tasks, the *control flow graph* generation and the code instrumentation. With this aim, OCELOT extends the `IASTVisitor` class in order to define specific visitors which implement the methods that will be called during the visit of specified *abstract syntax tree* element. Table 4.1 shows the main methods defined for the different concrete AST nodes.

Each visitor that has been used to instrument the code, needs to analyse the specific type of variable used in encountered instructions; CDT provides some useful utilities to facility this task: during the *abstract syntax tree* generation process, CDT performs a partial static code analysis and *binding*.

CDT also provides a very useful class, `ASTWriter`, which allows to transform the *abstract syntax tree* into the correspondent source code. This utility has been crucial, essentially because the code instrumentation process requires some AST modifications. The C code is written from the modified *abstract syntax tree* with a call to the `write()` method of a `ASTWriter` instance.

4.1.2 jMetal

jMetal stands for *Meta-heuristic Algorithms in Java* [14] and it is an object-oriented Java-based framework for mono/multi-objective optimization with meta-heuristic techniques. jMetal provides a rich set of classes which can be used as the building blocks of various techniques, taking the advantages of code-reusing and facilitating the implementation of new kind of operators or algorithms.

jMetal was chosen after a detailed evaluation of its characteristics; it is developed with the aim to be simple and easy to use, portable (is written in Java), flexible and extendible. Other reasons can be summarized in the following:

- **Simplicity and easy-to-use**. The base classes like `Solution`, `SolutionSet`, `Variable` and their operations are intuitive and, as a consequence, easy to understand and use. Furthermore, the framework includes the implementation of many meta-heuristics, which can be used as templates for developing new techniques;

- **Flexibility**. On the one hand, jMetal incorporates a simple mechanism to execute the algorithms under different parameter settings, including algorithm-specific parameters as well as those related to the problem to solve. On the other hand, issues such as choosing a real or binary-coded representation and, accordingly, the concrete operators to use, should require minimum modifications;

- **Portability**. The use of Java as a programming language allows to fulfil this goal;

- **Extendibility**. New algorithms, operators, and problems can be easily added. All specific problems inherit from the class `Problem`, so a new one can be created just by writing the methods specified by that class.

A UML class diagram representing the main jMetal components and their relationships is depicted in Figure 4.1.

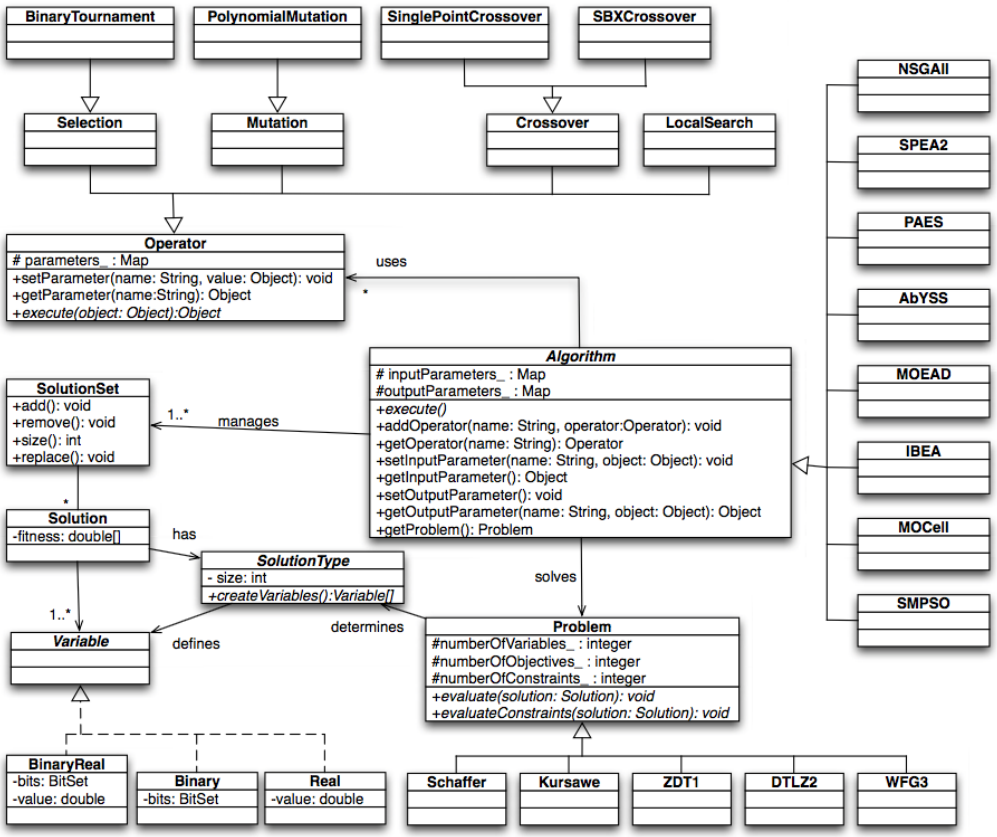


Figure 4.1: jMetal class diagram

The basic architecture of jMetal relies in that an `Algorithm` solves a `Problem` using one (and possibly more) `SolutionSet` and a set of `Operators` objects. In the context of evolutionary algorithms, population and individuals corresponds to `SolutionSet` and `Solution` jMetal objects, respectively.

Encoding of Solutions

Defining how to encode or represent the candidate solutions of the problem to solve is a crucial decision when using meta-heuristics. Obviously, representation strongly depends on the specific domain problem and also determines the operations that can be applied.

A `Solution` is composed by a set of `Variable` objects, which can be

of different types (binary, real, integer, etc.), plus an array to store fitness values. Each `Solution` has an associated type (the `SolutionType`). The mechanism allows to define the variable types of the `Solution` and creating them, simply by using the `createVariables()` method.

This encoding solution is very simple and easily allows to define more complex representations, mixing different variable types, that could be useful for particular problems.

Operators

Meta-heuristic techniques are based on modifying or generating new solutions from existing ones by means of the application of different operators. In jMetal, any operation altering or generating solutions inherits from the `Operator` class, as can be seen in Figure 4.1.

Besides allowing the definition of custom new operators, jMetal already incorporates a large number of operators, which can be classified into four different categories:

- `Selection`. This kind of operator is used to perform the selection procedures in many evolutionary algorithms. An example of selection operator is the *binary tournament*;
- `Crossover`. Represents the classic recombination or crossover operators used in evolutionary algorithms literature. Some of the included operators are the *simulated binary* crossover [12] and the *two-points* crossover for real and binary encodings, respectively;
- `Mutation`. Represents the mutation operator used in evolutionary algorithms. Examples of included operators are *polynomial mutation* [12] and *bit-flip mutation* for real and binary encodings, respectively;
- `LocalSearch`. Represents local search procedures.

Each operator defines the `setParameter()` and `getParameter()` methods, which are used in order to add and access to an specific operator parameter, e.g. the crossover probability. The operators can also receive their parameters by passing them as an argument when the operator object is created.

Problems

As said before, in jMetal, all the problems inherit from class `Problem`. This class defines two basic methods: `evaluate()` and `evaluateConstraints()`. Both methods receive a `Solution`, which represents a candidate solution for the specific problem; the first one evaluates it, and the second one determines the overall constraint violation of this solution. All the problems have to define the `evaluate()` method, while only problems having side constraints need to define `evaluateConstraints()`.

Algorithms

The last core class showed in Figure 4.1 is `Algorithm`, an abstract class which must be inherited by the meta-heuristics included in the framework and by all other ones that need to be implemented. In particular, the abstract method `execute()` must be defined; this method is intended to run the algorithm, and it returns a `SolutionSet` as result.

4.2 System architecture

A preliminary system design was the first necessary phase of the OCELOT building process. A good design phase is crucial because it allows to identify potential critical issues. In particular, in this first phase, the entire system has been decomposed into various subsystems, allowing parallel development and ensuring some architectural qualities such as abstraction, good insulation from change and replaceable implementation.

OCELOT is composed by 9 core subsystems which can be seen in Figure 4.2. This section describes in detail all OCELOT components.

4.2.1 Control flow graph generation

The control flow graph generator is an OCELOT core component. This process needs several classes to be accomplished; the most important ones are the following:

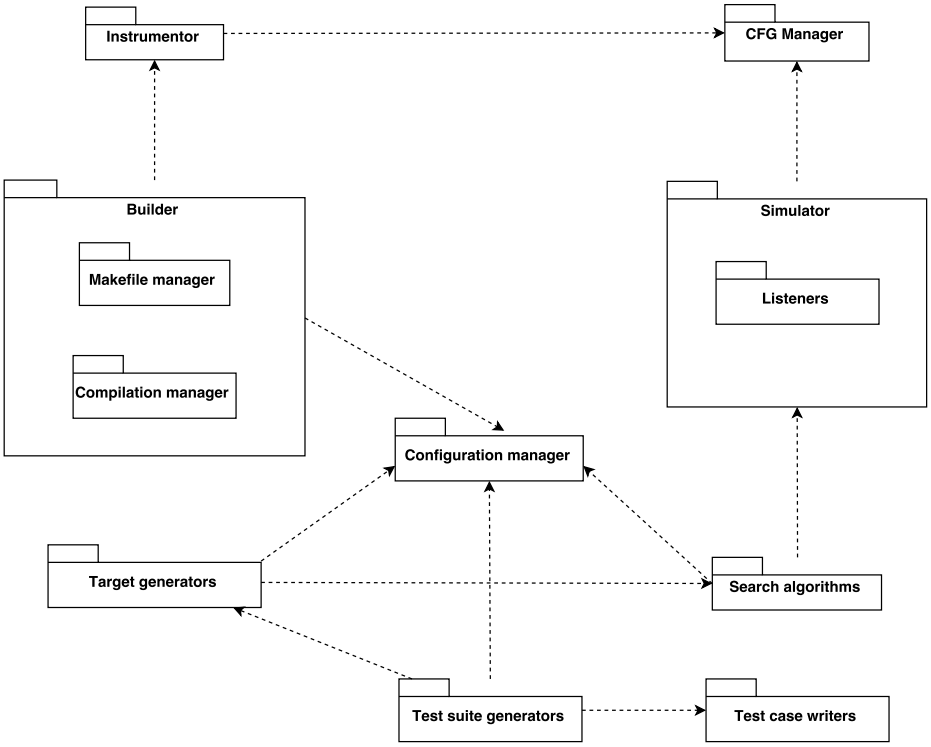


Figure 4.2: OCELOT package diagram

- `CFGVisitor`: an *abstract syntax tree* visitor which allows to generate the *control flow graph* of a function under test;
- `CFG`: the object which represents and implements a *control flow graph* This class uses the jGraph library to handle the graph.

OCELOT builds the *control flow graph* in an incremental way: the first step is a depth-first search on the *abstract syntax tree*; then the process continues with the creation of sub-graphs generated by the simpler instructions. Each sub-graph is identified by:

- a list of input nodes, which represents the graph entry nodes;
- a list of output nodes, which represents the graph exit nodes;
- additional lists of case and cycle nodes for `switch` and for `break`, `switch`, and `continue`, respectively.

```
1 if (a > 10) {
2     b = 11;
3 } else {
4     b = 10;
5 }
```

Listing 4.1: Code example

This process can be described considering the piece of code in Listing 4.1. The first step is the creation of the two sub-graphs for lines 2 and 4. The two resulting graphs will have the same lists of input and output nodes, i.e. the node itself. Table 4.2 shows this partial result.

List	**Sub-graph A**	**Sub-graph B**
input	b = 11	b = 10
output	b = 11	b = 10

Table 4.2: Partial result after the creation of the two sub-graph from the instructions *b=11* and *b=10*

Table 4.3 shows indeed the result after the visit of the entire control structure. The resultant graph is showed in Figure 4.3; obviously it may be used itself as sub-graph in an incremental way, from an another parent.

List	**Sub-graph C**
input	a > 0
output	A and B outputs [(b=11), (b=10)]

Table 4.3: Result from the entire *if* structure, given the two graphs A and B

A label describes the type of every edge in the graph. All edges are classified into four categories:

- `Flow`. An edge which is inevitably covered when the execution reaches its source node. It starts from a node with *outdegree* 1;

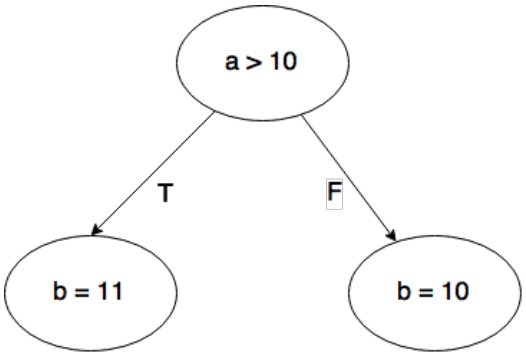


Figure 4.3: Final control flow graph for code fragment in Figure 4.1

- `True`. An edge which start from a branching node; its source node must have an outgoing edge label with `False`;
- `False`. An edge which start from a branching node; its source node must have an outgoing edge label with `True`;
- `Case`. An edge starting from a `switch` node; its source node has an *outdegree* greater than 0 and it has only `Case` outgoing edges. In this case, the label reports also information about target node (a `Case` expression).

4.2.2 Code instrumentation

Code instrumentation is the most critical task in OCELOT; this is due to the huge variance of possible cases, combinations and types to consider. Code instrumentation provides a way to:

- trace all non-trivial events (i.e. all decisions taken in correspondence of a predicate);
- calculate *branch distance* values (section 2.3.3 for more details).

First of all, the following C functions needed to instrument the code have been defined and implemented:

- `_f_ocelot_trace`: adds a non-trivial event to the correspondent list;

Function	Description
`_f_ocelot_trace`	Traces an events and records distances
`_f_ocelot_trace_case`	Traces an event which refers to a case instruction. It records distance from true branch and a flag which indicates that the branch has been covered
`_f_ocelot_eq_numeric`	Calculates the distance from the fulfilment of condition $a == b$
`_f_ocelot_gt_numeric`	Calculates the distance from the fulfilment of condition $a > b$
`_f_ocelot_ge_numeric`	Calculates the distance from the fulfilment of condition $a \geq b$
`_f_ocelot_neq_numeric`	Calculates the distance from the fulfilment of condition $a \neq b$
`_f_ocelot_and_numeric`	Calculates the distance from the fulfilment of condition $a \wedge b$
`_f_ocelot_or_numeric`	Calculates the distance from the fulfilment of condition $a \vee b$

Table 4.4: Functions used in code instrumentation process in order to trace non-trivial events. For clarity, this table shows only functions for numeric expressions

- `_f_ocelot_XXX_numeric`: calculates the distance for a given predicate (where `XXX` corresponds to the various equals, greater, greater than and so on) which handles numeric expressions;
- `_f_ocelot_XXX_pointer`: calculates the distance for a given predicate which handles numeric pointer expressions.

Table 4.4 shows the main functions used in OCELOT to trace non-trivial events and to record the distances. The functions for *less than* and *less-equal than* operators have not been implemented because they can be obtained from the *greater than* and *greater-equal than* operators.

The entire process continues with the visit of the *abstract syntax tree*; each branching condition is replaced by a call to the proper function which traces the relative event. For each event, the following field are recorded:

- a value choice, 0 for `false` and 1 for `true`;
- distance from branching node `true` branch;

- distance from branching node `false` branch.

So a tracing function needs three parameters: the value choice and two distances, from `true` and `false` branch, respectively. The original condition is used to trace the choice; the proper function traces the distance relative to original expression (so relative to `true` branch), while the distance from `false` branch is traced by the appropriate function for the expression made by the negation of original condition. The `switch` instruction requires a more complex instrumentation; in this case, n events are recorded, where n is the number of `case` instructions.

An example of code instrumentation output is shown in listings 4.2 and 4.3; listing 4.2 shows the original piece of code before the instrumentation process; listing 4.3, indeed, shows the final output.

```
1 if (a == b) {
2     b = 11;
3 } else {
4     b = 10;
5 }
```

Listing 4.2: Original code

```
1 if (_f_ocelot_trace(
2     a == 10,
3         _f_ocelot_eq_numeric(a, 10),
4             _f_ocelot_neq_numeric(a, 10))
5 ) {
6     b = 11;
7 } else {
8     b = 10;
9 }
```

Listing 4.3: Instrumented code

With this information about control flow execution, the tracing function knows which decision nodes have been executed and also knows the correspondent outcome, i.e. the relatives as branch distances. Obviously, the entire process does not alter the behaviour of the program, but it only observe it.

4.2.3 Library compilation

From the instrumented code, it is possible to compile the C library which will be used by OCELOT through Java Native Interface (JNI). Some *ad-hoc* functions have been defined in order to allow the communication between Java code and the native library. The component which handles the *build* process consists in two distinct parts:

- Makefile generator: deals with the automated generation of the *makefile*, according to the underlying operative system;
- Compilation manager: defines some utility functions depending from the used C compiler; it deals to perform the *build*.

Multi-platform support has been the most challenging problem to address in this phase. This problem has been solved with a Bridge Design Pattern [20] (in Figure 4.4) that is an object structural pattern which allows to decouple an abstraction from its implementation. In OCELOT, a specific abstract class, `JNIMakefileGenerator`, has been defined: this class defines several abstract methods that must have implemented by every subclasses (see section 4.4 for more details). Several classes which implement the support to the *makefile* generator for a specific operative system (Mac OS X, Linux and Windows), have been implemented. When OCELOT has to build the library, a concrete instance of `Builder` class performs all necessary operations and generates the specific *makefile* from a concrete instance of `JNIMakefileGenerator` provided by the caller.

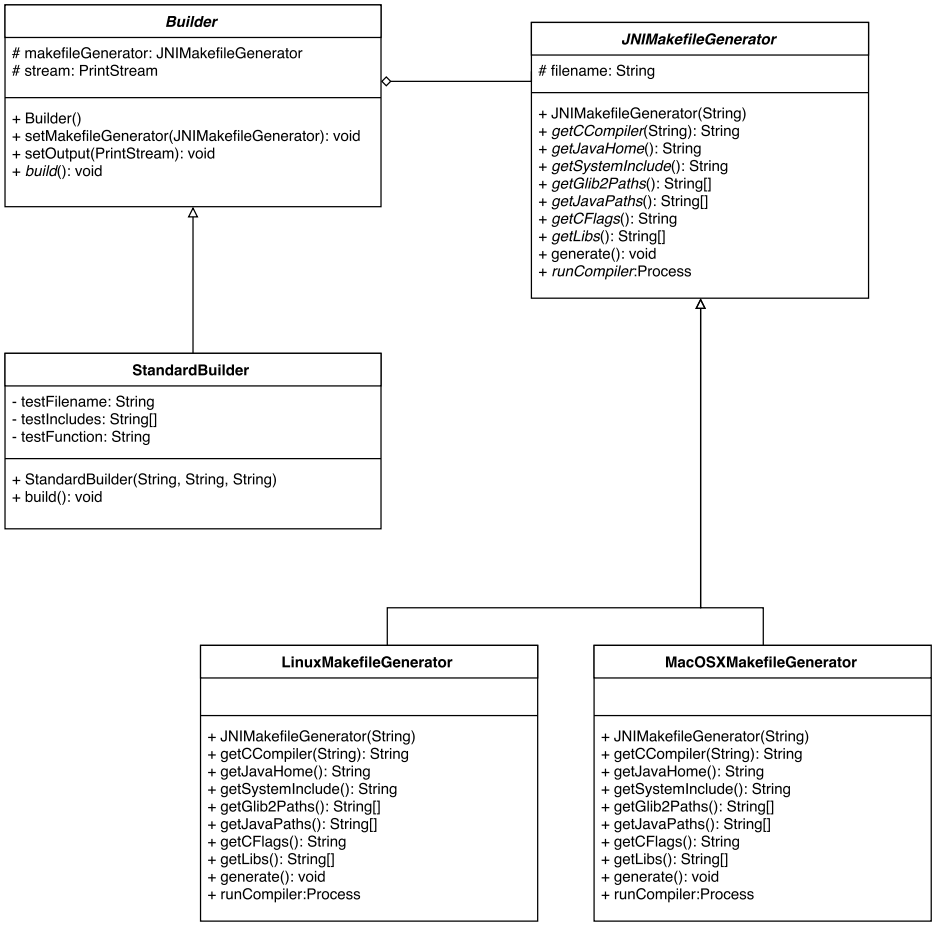


Figure 4.4: Bridge design pattern

4.2.4 Simulation

The library generated as above, which contains the instrumented code, now could be called by the Java code through Java Native Interface. OCELOT, with a single method call, is able to execute the function under test, tracing all non-trivial events. A crucial component that, from the *control flow graph* and from the list of traced events, allows a simulation the execution, has been developed. Since the simulation is performed on the graph, this technique allows to perform all needed calculations without any impact on the source code.

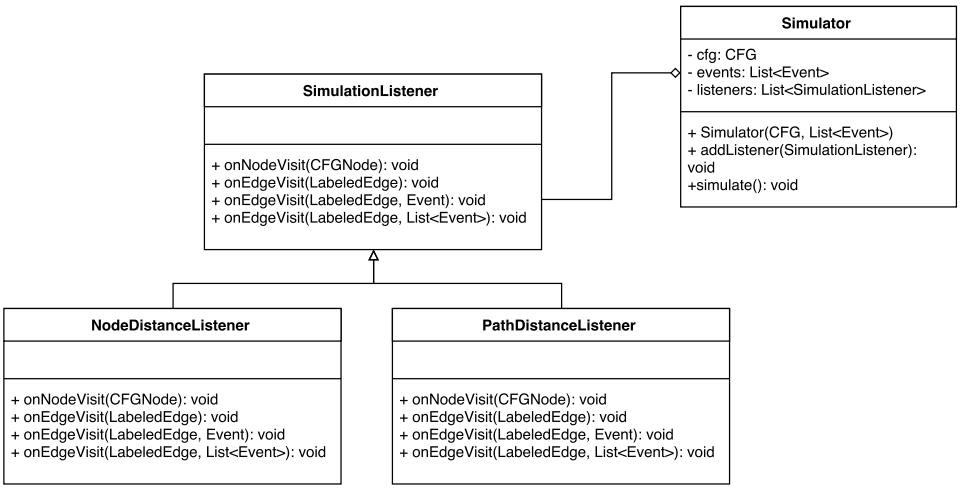


Figure 4.5: Observer Design pattern

In order to implement a solid and expansible system, an *Observer Design Pattern* [20] has been used to handle the simulation mechanism (Figure 4.5). The *Observer* pattern is used to define a one-to-many dependency between objects; it allows an automated notification to all dependent, when the state of the observed object changes. This approach allows an easy way to develop new techniques: a new implementation of the `SimulationListener` interface will be sufficient. The simulator notifies to all linked listeners the following events:

- visit of a node;
- visit of a `Flow` edge;
- visit of a `True` or `False` edge;
- visit of a `Case` edge.

OCELOT provides several kind of useful listeners, such as:

- `NodeDistanceListener`: it calculates the *branch distance* and the *approach level* between the control flow and the target node;

- `PathDistanceListener`: it calculates the *branch distance* and the *approach level* between the control flow and a path which is the current target;
- `DominatorListener`: it calculates the *branch distance* and the *approach level* between the control flow and a target edge, using the concept of dominance (see section 3.3.1);
- `CoverageCalculatorListener`: it calculates some different coverage metrics for a given control flow.

4.2.5 Solution searching

The parts of the system described before, are used by a component that deals with the search of those parameters which lead to reach the coverage for a certain target. As said, the literature about search based testing data generation has widely exploited global search algorithms, such as *simulated annealing* or *genetic algorithms* (see section 2.2); for this reason, the *genetic algorithms* has been the first search techniques implemented in OCELOT. However, the tool architecture easily allows the implementation of other ones.

In the contest of structural testing, the aim is to minimize an objective (or fitness) function which refers to the distance from a specific element (a node, a path or a branch) of the control flow graph that corresponds to the target of the current search. For example, in order to reach a particular node in the object test, the fitness function, which is calculated by combining the branch distance with the approach level, indicates the distance between the current execution flow and that specific node; obviously, the target will be covered by an input vector for the object test, only if the value of the fitness function reaches the zero. For this specific problem, OCELOT represents each solution as an array of real numbers, using the `ArrayRealSolutionType` type provided by jMetal [14]; in this representation, each element of this vector naturally refers to a single input argument for the test object. Some little changes in this representation are necessary for pointers handling (see subsection 4.2.10).

As said above, jMetal framework [14] has been used in order to speed up the genetic algorithm implementation. In particular, each approach needs to extend three fundamental jMetal classes:

- `Experiment`: it is the main class that instances a `Setting` and a `Problem` object, allowing the execution of the correspondent algorithm;
- `Settings`: it instances the specific algorithm, based on the `Problem` dynamic type, and it sets its setting parameters and operators;
- `Problem`: it executes the native C source code with an input vector that refers to a specific solution and it performs the relative simulation on the control flow graph. In the `evaluate()` method, this class evaluates each solution, calculating the correspondent fitness function.

4.2.6 Test suite generation

As described in section 2.3.3, every single target approach needs to select some intermediate objectives, in order to reach the maximum available coverage for the test object; for this reason, a component which defines these intermediate objectives is crucial. Since, usually one or more intermediate objectives are addressed by a test case, this component is closely linked to the component that generates them.

Regarding single target approaches, OCELOT currently implements several different methodologies for selecting the intermediate objectives:

- a *naive* selection of all branches;
- a smarter *memory-based* selection approach;
- a selection of each McCabe independent paths.

In the first case, all branches of the control flow graph are targeted, one by one, by a distinct search process; for each successful search, the algorithm generates a test case (Algorithm 11).

In the second case, discussed in chapter 3 (see Algorithm 9), the algorithm removes, from the set of remaining targets, all the branches covered by the

Algorithm 11: *Naive* algorithm

Input: P: target program; B: set of all branches in P
Output: TS: a test suite
$TS = \emptyset$
foreach $b \in B$ **do**
  $t \leftarrow search(b)$
  $TS \leftarrow TS \bigcup \{t\}$
end
return TS

selected test case whenever it generates a specific input for a given target branch.

At last, in the third case, the algorithm selects all McCabe linearly independent paths (Algorithm 12), trying to find a solution vector for each of them.

The component which generates the test cases, can use one or more selectors, combining them with the aim to increase the overall coverage. For example, the approach proposed by Scalabrino [51] combines the basis paths selector with the memory-based selector, if the first does not achieve the full coverage for the test object.

OCELOT also implements a test suite generator for the MOSA many-objective approach (described in section 3.2). In the MOSA algorithm, the component which generates the test cases is closely linked to the UPDATE-ARCHIVE function (Algorithm 13) that stores the set of solutions which are

Algorithm 12: Algorithm which uses the basis paths

Data: $CFG = (V, E)$ control flow graph
Result: TS: a test suite
$TS = \emptyset$
foreach $b \in B$ **do**
  $t \leftarrow search(b)$
  $TS \leftarrow TS \bigcup \{t\}$
end
return TS

Algorithm 13: Update Archive

```
Input: A set of candidate test cases T
Result: An archive A
A ← ∅
for b_i ∈ uncovered branches do
    t_best ← ∅
    covered ← ∅
    for t_j ∈ T do
        score ← objective score of t_j for branch b_j
        if score == 0 then
            t_best ← {t_j}
            mark b_i as covered
            for b_k ∈ uncovered branches do
                if score == 0 and b_j is uncovered then
                    mark b_j as covered
            end
    end
    if t_best ≠ ∅ then
        A ← A ⋃ t_best
end
```

candidate to become test cases. As said, since the MOSA algorithm has been developed for Java test objects (section 3.2), the original implementation of the UPDATE-ARCHIVE function considers also the length of the test cases. However, regarding the C language, the length is an useless parameter, since each test case is simply composed by a call to the function under test and by an assertion. So, the correspondent OCELOT UPDATE-ARCHIVE function (Algorithm 13) works a little differently: for each uncovered branch b_i, it looks for a solution which has an objective score equals to zero; then it marks b_i as covered and it looks for other branches which might have been collaterally covered while not explicitly targeting. When the algorithm stops, the set of solutions stored in the *archive* will form the resulting test suite.

4.2.7 Test cases definition

Another important OCELOT component deals with the automated definition and writing of test cases. With this aim, the tool uses the CDT library in order to create an *abstract syntax tree* which will be translated in C source code using the same library. The tool has been designed in order to guarantee extendibility; with this aim, a Bridge design pattern [20] has been used to select a specific implementation of this component. Each test case is composed

by the following parts:

- a call to the function under test, with the appropriate parameters obtained from the evolutionary search;
- the oracle check (assertion);

As said, although efficient techniques for the automated oracle definition may be possible from precise models and checkable specifications, however usually the system behaviour must be checked by a human. For this reason, that component assumes that the function has a correct behaviour and it only inserts the assertions regarding the function return value.

4.2.8 Configuration handler

OCELOT handles the principal configuration parameters with a proper configuration file. These parameters refers to:

- genetic algorithm parameters, like the population size or the crossover and mutation probability;
- information about the test object and the function under test.

Figure 4.6shows a configuration file example.

4.2.9 A new mutation operator

In OCELOT, a new mutation operator, based on the *seeding* concept [17], has been implemented. This operator, with a given probability, simply mutes a single *gene* of the chromosome into a random numeric constant of the function under test. With this aim, OCELOT defines and implements a specific visitor, called `ConstantCheckerVisitor`, that parses the *abstract syntax tree* and registers all the numeric literals.

That operator works results very efficient when it encounters some predicates with conditions that are particularly difficult to satisfy. For example, considering the piece of code in figure 4.4, it is easy to understand that, for a stochastic algorithm like the genetic one, the precise double value 0.0 (line

```
population.size:100
evaluations.max:50000
crossover.probability:0.8
mutation.probability:0.2
threads:1

test.debug:false
test.ui:false
experiment.output.folder:./outputs/
experiment.results.folder:./experiments/
experiment.results.print:true
experiment.runs:1

suite.generator:McCabe
suite.minimizer:AdditionalGreedy
suite.writer:CUnit

test.basedir:testobject/
test.filename:test.c
test.includes:folderA/;folderB/

test.function:test_me
test.target:Flow,True,Flow
test.parameters.ranges:-1000:1000 -1000:1000 -1000:1000
```

Figure 4.6: OCELOT configuration file example

1) is hard to be exactly reached. In this case, the new operator has a certain probability to mute a, assigning it the exact value 0.0.

This approach clearly facilities the coverage of a particularly challenging condition, but on the other hand it could introduce an annoying noise that

```
if (a == 0.0)
    a++;
else
    a -= 12;
```

Listing 4.4: Example of condition which is particularly difficult to satisfy

disturbs the search with useless literals. So, it is important to tune the probability of this kind of mutation in an appropriate way.

4.2.10 Pointers handling

The pointer handling is an annoying and challenging problem in structural testing. Since in that case, the traditional distance approach compares the memory locations, it is hard to define a guidance for the search towards the appropriate test data. In order to support the pointers usage, some improvements in the genetic algorithm, in the code instrumentation and in the definition of fitness function have been necessary.

Genetic algorithm In the OCELOT standard genetic algorithm, a chromosome is represented by an array of real number with an `ArrayRealSolutionType` object; in this representation, each element refers to a single function parameter. In order to support the usage of pointers, OCELOT defines a slightly different chromosome representation. In particular, three arrays are used as follow:

- an array of real number for non-pointer variables;
- an array, with a standard size specified in the configuration file, for each pointer;
- an array that associates each pointer to a previously defined array.

The following example explains this representation. Given the function

```
test(int a, int b, int* c, int* d, int* e)
```

a chromosome like the following is possible:

$[10, 22]$	`a` and `b` values
$[0, 1, 22, 3]$	array #0
$[23, 2, 2, 65]$	array #1
$[22, 56, 3, 3]$	array #2
$[0, 0, 1]$	pointer association

The above representation will refer to the following association:

`a` = 10
`b` = 22
`c` = pointer to the array $[0, 1, 22, 3]$
`d` = pointer to the array $[0, 1, 22, 3]$
`e` = pointer to the array $[23, 2, 2, 65]$

It is important to notice that, it this particular representation, both crossover and mutation operators, consider each array individually. This means that, for example, when a crossover operation occurs, it performs a single operation for each chromosome part, i.e. one crossover for the real values array, one for each pointer array and one for the array which associates the pointers to their definition.

Instrumentation In order to support pointers, also the instrumentation process has been slightly modified: in the C source code, a matrix will record all the arrays of a chromosome. According to the previous example, OCELOT creates a matrix like the following:

0	1	22	3
23	2	2	65
22	56	3	3

Then it assigns each pointer to a correspondent row of this matrix, according to the last chromosome part which associates a pointer to its definition.

Fitness function In order to support pointers usage, a new definition of branch distance for two pointers is necessary. First, with some basic arithmetical operations it is possible to know the row index of the two pointers; then, these indexes are simply used in a normal comparison. Considering the following piece of code.

```
1 if (c == d)
2     if (d == e)
3         return true;
```

According to the previous example, the variable `c` has a row index equal to 0; `d` also has a row index equals to 0, when `e` has an index equal to 1. So, calculating the distance about the condition at line 1, the resulting distance value is 0, while obviously the distance value for the condition at line 2 is 1.

4.3 Execution process

The execution process of OCELOT consists in two main phases:

- Build: this process first identifies the function under test from the configuration file; then it performs the code instrumentation task, recording all the choices done for each control structure. At last, the OCELOT standard libraries are imported and the resulting code is compiled in order to be used through JNI;
- Execution: this phase allows the simulation of the program execution through the library previously compiled. Here, the genetic algorithm is executed with the aim to search for an input vector that allows the coverage of a given target.

4.4 Build

As said, the build process allows the compilation of the library that contains the function under test. The executable class, `Build`, first instantiates a `StandardBuilder` object with all configuration parameters provided by the `ConfigManager` class. After that, the next step involves the makefile generation, according to the underlying operative system. OCELOT implements an extension of the abstract class `JNIMakefileGenerator` for each supported operative system; each concrete class provides some appropriate parameters, like the HOME Java directory, or the path of GLib2 library. With the call of the method `build()`, the original source code is instrumented and compiled. Two important visitors are used in this task:

- `InstrumentorVisitor`: it allows the concrete code instrumentation, representing the most crucial and critical visitor. It defines and implements all the `visit` methods which are used to construct the *abstract syntax tree*. During this process, the original source code is instrumented with the various `_f_ocelot_` functions in order to trace the execution flow and the informations relative to the distances;
- `MacroDefinerVisitor`: it generates the proper macro which allows the call of the instrumented and compiled C code through JNI.

After this process, a file containing the instrumented code is saved.

4.5 Execution

The following step in the OCELOT execution process is the execution of a particular test suite generator that is specified in the configuration file. The test suite generator:

- selects the set of target;
- runs the genetic algorithm for each target; for the many-objective approach it runs the algorithm considering all targets simultaneously;

- eventually it minimizes the generated test suite.

Currently, the parameters of the generated test cases are printed on the output console, with some useful benchmarks (in particular the time, the test suite size and the coverage benchmark).

Chapter 5

Case of study

Previous chapters described how search-based testing has been used to automatically generate test data for structural testing. Both methodologies have been evaluated with the aim to increase the effectiveness of OCELOT, the previously proposed tool. The aim of this chapter is to provide a detailed description of the empirical study which has been performed, in order to compare the two novel methodologies, detailed in chapter 3, with the approach proposed by Scalabrino [51], based on the McCabe independent paths. The case study has been conducted using OCELOT.

5.1 Aim and context

The main aim of this empirical study is to compare the effectiveness of the two novel algorithms discussed (MOSA and CDG based algorithm) against the approach proposed by Scalabrino [51] in his master thesis. Both approaches have been implemented in OCELOT with the aim to increase its performance, both in term of coverage and test suite size. The basis paths based methodology can be resumed as follow:

- it calculates the basis paths for the object under test;
- it performs a search for each basis path with the path itself as target;

- if the maximum coverage has not been achieved, it performs a single-target search for each uncovered branch, in order to increase as much as possible the overall coverage; otherwise, the algorithm terminates.

In order to assess the validity of this approach, Scalabrino performed an empirical study which compares the achieved branch coverage and the obtained test suite size between random testing [13] and the proposed basis paths technique. The results of this study suggest that this approach reaches higher coverage than random testing (about 7.1% on average) with a smaller number of test cases generated (54.9% less).

Test subjects Five functions have been used as test subjects. The details of the test subjects are recorded in Table 5.1. Obviously, the functions chosen are not trivial for automated test data generation. `triangle` is a well-known triangle classification function, often used as a benchmark in several automatic test data generation. `gimp_rbg_to_hsl_int` is a function that projects a color from RGB to HSL space. This is the hardest function to entirely cover, essentially because it has a lot of nested `if` and also because some variables, derived from those in input, are involved in several predicates. The function `gimp_rbg_to_hsc4` is quite similar to previous function; it projects a color from RGB to HSV space. The `cliparc` function is the largest test object used in the empirical study. Despite its size, this function has not particular issues but it presents a *sanity check* which limits the maximum coverage that can be achieved to about 95%. At last, the function `Csqrt` allows to calculate the square root of a complex number. This function is not too complex but it presents a comparison with the double value zero as particular issue.

5.2 Design

The goal of the empirical evaluation is to assess the effectiveness and the efficiency of MOSA and CDG based approach, in comparison with the basis paths approach [51] implemented by OCELOT.

Function	Lines of code	Branches	Cyclomatic complexity
`triangle`	21	14	7
`gimp_rbg_to_hsl_int`	58	14	7
`gimp_rbg_to_hsc4`	62	18	9
`cliparc`	136	64	32
`Csqrt`	26	6	3

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

In particular, the research questions to be answered by the empirical study are therefore as follows:

RQ 1: *What is the coverage achieved by MOSA and CDG based algorithm versus basis paths technique?*

RQ 2: *What is the size of resulting test suite obtained by MOSA and CDG based algorithm versus basis paths technique?*

RQ 3: *What is the rate of convergence and the cost of MOSA and CDG based algorithm versus basis paths technique?*

RQ 1 aims at evaluating if either MOSA and CDG bases technique are able to reach higher branch coverage, if compared to the approach based on McCabe's paths; **RQ 2** indeed, aims at evaluating the test suite size obtained by the three approaches, in order to reach the same coverage. At last, **RQ 3** focuses on the cost required by McCabe, MOSA and CDG based techniques.

Metrics

In order to compare the three techniques, branch coverage has been used as a measure in order to address **RQ 1**; the coverage achieved by each single technique for a given function is computed as the number of branches covered by the resulting test suite, divided by the total number of branches. **RQ 2**

Parameter	Value
Population size	100
Crossover rate	0.8
Mutation rate	0.2
Constant Mutation rate	mutation rate/20
Search budget (evaluations)	30.000

Table 5.2: Parameter settings

trivially refers to the resulting test suite size as measure of effectiveness. Finally, **RQ 3** uses the number of evaluations performed by each technique during the overall search process, as measure of efficiency.

Experimental Setup

For each function and for each strategy, the following metrics have been collected, both for each single test case and as a cumulative result:

- the time taken by the search;
- the resulting test suite size;
- the achieved branch coverage;
- the number of evaluation of a solution performed by the search;

The search stops when either full branch coverage or the maximum number of evaluations is reached. Obviously, the entire process of data collection is full automated.

Several parameters, like population size or operator probability, tune the performance of a genetic algorithm. Despite Harman et. al. [27] refer to other crossover, selection and mutation operators for the implementation of CDG based algorithm, the empirical study presented in this chapter uses the same operators for all techniques, in order to exercise them with the same conditions: the *SBX Crossover* as crossover operator, the *Binary Tournament* as selection operator and *Polynomial Mutation* combined with the *Constant Mutation* operator discussed in section 4.2.9, as mutation operator. It is

Function	McCabe	MOSA	CDG based
`triangle`	0.929	0.929	0.914
`gimp_rbg_to_hsl_int`	0.929	0.929	0.907
`gimp_rbg_to_hsc4`	0.666	0.666	0.666
`cliparc`	0.945	0.945	0.948
`Csqrt`	1	1	1

Table 5.3: Branch coverage achieved by each approach

important to notice that no analysis about the performance comparison of different operators, in order to determine their effectiveness, has been performed in this study. The other values for the important search parameters are listed in Table 5.2. Due to the stochastic nature of the genetic algorithms, the experiments were repeated 10 times and the results reported are averages over these 10 runs.

5.3 Results

This section shows the results of the empirical study, divided by the correspondent research question.

RQ 1 Branch Coverage levels Table 5.3 summarizes the branch coverage values achieved by each approach. The same values, combined with the average test suite size, can be seen in the bar charts of Figure 5.1. It is easy to observe that the average coverage achieved by the three different algorithms is quite the same for all five functions under test; so, regarding the overall branch coverage, MOSA and CDG based algorithm seem not to introduce significant improvements. Since the McCabe approach shows an improvement about 7% on average coverage [51], compared to a random search, it is reasonable to say that MOSA and CDG based algorithm have a quite similar effectiveness improvement.

In summary, it is possible to conclude that both McCabe, MOSA and CDG based approach, achieves quite the same amount of branch coverage.

Function	McCabe	MOSA	CDG based
`triangle`	10.5	6.8	5.8
`gimp_rbg_to_hsl_int`	11.2	5	4.9
`gimp_rbg_to_hsc4`	32	2.8	2.5
`cliparc`	43.5	16.6	14.6
`Csqrt`	4	3.2	3.4

Table 5.4: Average test suite size using different approaches

RQ 2 Test Suite size Table 5.4 summarizes the average test suite size using different approaches. Figure 5.1, which has been seen above for the branch coverage results, shows the same values as bar charts (the dark grey bars on the right). As can be seen, for every test subject, the average test suite size generated by both novel algorithms was significantly smaller than the McCabe approach one. It is important to notice that, for every test subject, the resulting average test suite size from the two novel algorithms was smaller than the subject's cyclomatic complexity. So, both MOSA and CDG based approaches perform better compared to the McCabe technique: MOSA gives the best result in just 1 case, while CDG based approach performs better in the remaining 4 cases. Finally, from the results analysis reported in Table 5.3, it is possible to conclude that the MOSA algorithm generates, on average, about 52.74% smaller test suites, while the CDG based algorithm, about 54,94% smaller ones.

In summary, according to the results reported above, it is possible to conclude that both MOSA and CDG based algorithms produce smaller test suites than the McCabe approach, without sacrificing the achieved branch coverage. However, the CDG based technique seems slightly better than MOSA regarding the test suite size.

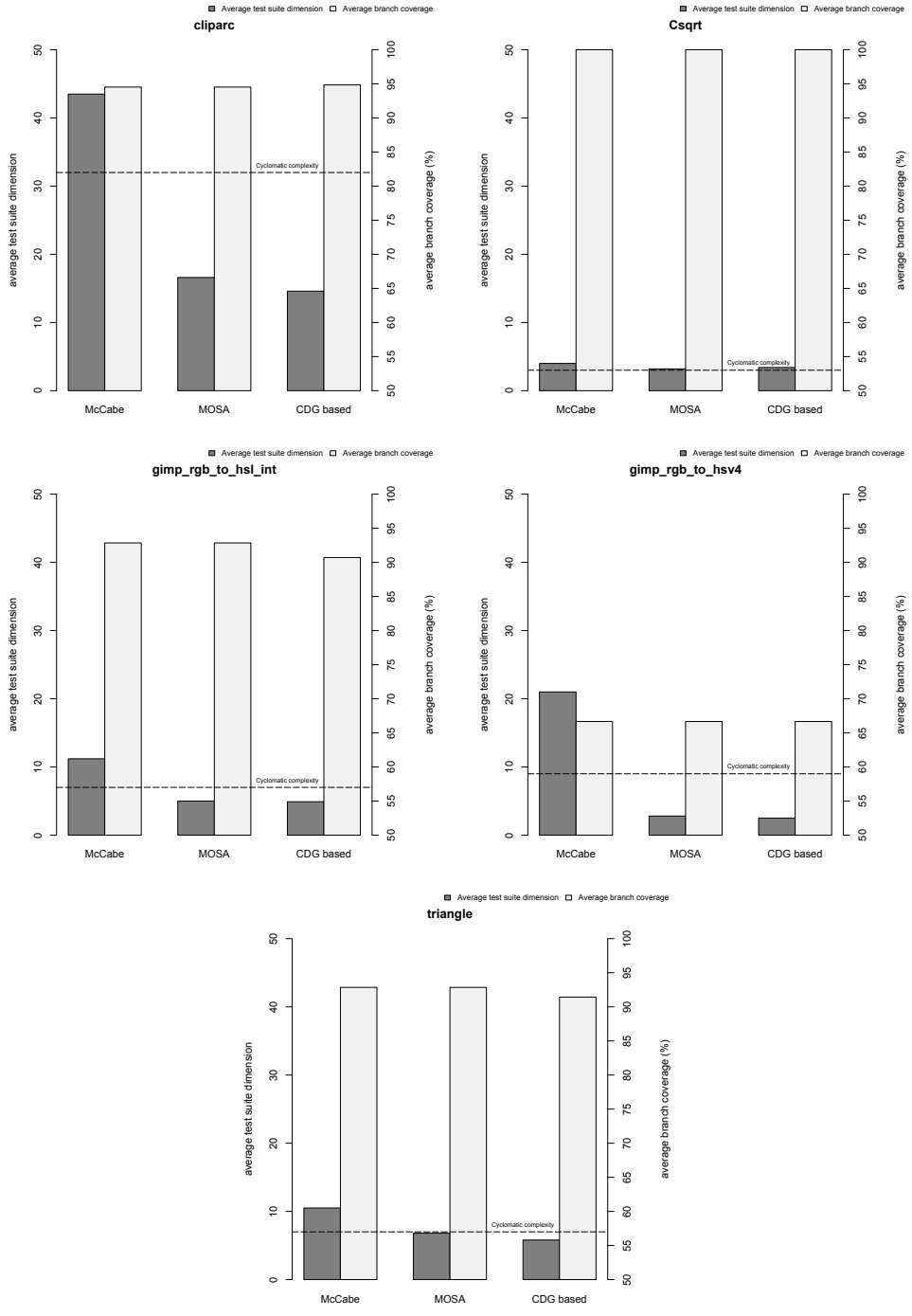


Figure 5.1: Average branch coverage and test suite size achieved using different approaches

Function	McCabe		MOSA		CDG based	
	T(s)	Evaluations	T(s)	Evaluations	T(s)	Evaluations
`triangle`	5.5	196.766	2.7	26.351	0.9	76.532
`gimp_rbg_to_hsl_int`	20.5	259.252	5.2	6.054	3.1	66.280
`gimp_rbg_to_hsc4`	83.2	841.764	26.8	340	18.2	241.009
`cliparc`	453.7	1.526.544	44.6	79.420	30.1	133.753
`Csqrt`	0	1.409	0	1.086	0	1.467

Table 5.5: Time for performing the search and number of iterations needed to reach the maximum coverage for each algorithm

RQ 3 Cost required Table 5.5 summarizes both the time taken and the number of individual evaluations performed by each approach, in order to reach the maximum coverage and stop the search. To be thorough, this table reports also the cost for each algorithm expressed in seconds; otherwise, this study uses, as a metric to determine the cost and the convergence rate of each algorithm, the number of evaluations of a single candidate solution, performed during the search process. A solution evaluation is simply the calculation of the fitness function for that solution. Since the genetic algorithm process (see subsection 2.2.2) consists in a cycle which involves an evaluation of each individual, the number of evaluations performed can be properly used as measure of convergence rate. For this reason, Figure 5.2 refers only to the number of evaluations performed; these graphs show the increment of coverage when the number of evaluations grows up. It is clear that the MOSA algorithm uses less evaluations to reach the same coverage than the other two algorithms. Since the range of the evaluations is particularly large, the values on the abscissa axis are expressed in a logarithmic scale. It is important to notice that each value has been recorded when a single test case is added to the test suite by the corresponding algorithm. From the values reported in Table 5.5, it is possible to say that the CDG based algorithm reaches its maximum coverage with, on average about 58.81% less evaluations compared to the basis paths approach, while the MOSA algorithm needs on average about the 80.39% less evaluations.

According to this analysis, it is possible to conclude that the MOSA algorithm converges more quickly than both McCabe and CDG based techniques.

5.4 Analysis

The empirical study has been performed with the precise aim to introduce some improvement in branch coverage, test suite size and convergence rate, to the current approach used in OCELOT.

In terms of test suite coverage, the three techniques achieve similar average results. So, it is possible to conclude that the two novel approaches implemented in OCELOT do not provide a significant improvement about coverage against the technique based on the basis paths. However, remarkable improvements can bee seen evaluating the test suite size and the convergence rate.

In terms of test suite size, the basis paths approach is the worst; although this is not surprising, given that it generates a test case for each single-target search performed, even if that search does not lead to increase the coverage. Both MOSA and CDG-based algorithms generate a test suite more than 50% smaller, compared to the test suite generated by the basis paths approach. Although both approaches have very similar performance, the CDG-based technique produces, on the average of the five test objects, about a 6.64% smaller test suite.

At last, in terms of convergence rate, the clear winner is the MOSA algorithm. As seen previously, both MOSA and CDG-based technique converge faster than the basis paths approach; however, since MOSA is a many-objective algorithm able to search simultaneously for more targets, it is not surprising that it needs less evaluations to reach the same coverage. In details, it needs about 64.58% less evaluation than the CDG-based approach.

According to the analysis reported above, in summary, it is possible to conclude that there are no significant differences in branch coverage for the three presented approaches; however, MOSA converges much more quickly than the others two algorithms, while it is possible to affirm that the CDG based algorithm performs slightly better than MOSA and much better than the basis paths approach, in terms of resulting test suite size.

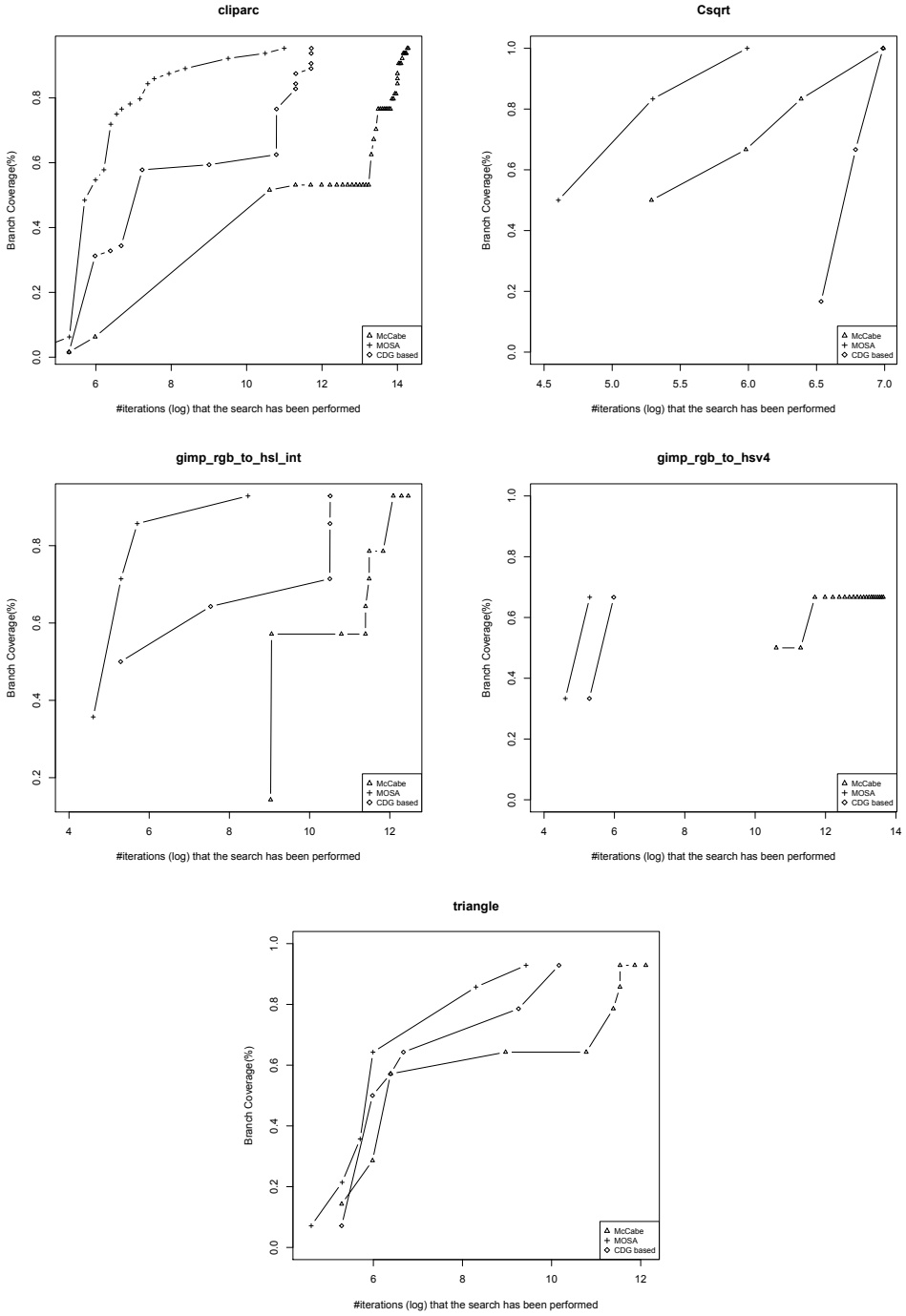


Figure 5.2: Coverage of each approach relative to the number of performed evaluations

Chapter 6

Conclusions and Future Work

This work aims to implement in OCELOT new methodologies for automated test case generation recently introduced in literature, in addition to the current approach based on the McCabe's linearly independent paths [51], with the final aim to improve the effectiveness and the efficiency of the tool, in particular in terms of coverage and test suite size.

A preliminary investigation of the recent literature has been performed in order to choose the most promising methodologies to implement in OCELOT. This process led to the identification of two approaches in particular. The first of them [47] follows the line traced by the *whole suite* approach [18] which demonstrated that considering all target simultaneously is superior to the traditional single-branch approach. However, instead of aggregating multiple objectives into a single value, as in the *whole suite* approach, the proposed algorithm, called MOSA, proposes to reformulate the branch coverage problem directly as a many-objective optimization one. In particular, this algorithm introduces a new preference criterion that allows to determine an order of preference over non-dominated test cases, allowing to overcome a huge limitation that had braked the adoption of multi-objective algorithms in the context of search based software testing. The second methodology [27] introduces a new formulation of the search based structural test data generation problem, in which the goals is to maximize the coverage, while simultaneously minimizing the number of test cases. With this aim, it focuses

on control dependence graph analysis in order to define a secondary fitness function that denotes the amount of structural collateral coverage achieved.

With the aim to evaluate the performance of OCELOT, using the different methodologies, all the approaches have been evaluated through an empirical study which compares them in terms of coverage, test suite size and convergence rate. Regarding the branch coverage, the three methodologies reach quite the same amount of coverage, so the two novel techniques do not introduce an improvement in relation to the current OCELOT approach. However, more interesting results come out observing the resulting test suite size and the convergence rate. The study demonstrates that both the MOSA and the CDG based algorithm generate a smaller test suite than the basis paths one, on the order of more than 50%. On average, the CDG based approach performs slightly better than MOSA. Regarding the convergence rate, MOSA is the clear winner: although the CDG based algorithm needs about 58% less evaluations than the basis approach to converge to its maximum coverage, MOSA clearly outperforms it (it needs about 64% less evaluations).

There are different directions to investigate for future work. First, it could be possible to introduce some elements from the CDG based algorithm in the many-objective approach. For example, a preliminary analysis of the control dependence graph could allow to restrict the set of target branches used by the algorithm, avoiding to consider those branches which will be inevitably covered while targeting other ones. Second, the introduction of different search algorithms (e.g. the Alternate Variable Method) could be considered with the aim to increase the coverage. At last, since it is widely recognized that the premature convergence to a sub-optimal solution inhibits the effectiveness of the evolutionary test case generation bases on genetics, the many-objective approach could be improved by integrating a mechanism that allows to maintain diversity in the population. Several works have been conducted in order to address this phenomenon, called *genetic drift* [49]. In particular, Kifetew et. al. [31] propose a technique for the orthogonal

exploration of the search space that enriches the diversity in the population by adding individuals in orthogonal directions.

Bibliography

[1] Wasif Afzal, Richard Torkar, Robert Feldt, and Tony Gorschek. Prediction of faults-slip-through in large software projects: an empirical evaluation. *Software Quality Journal*, 22(1):51–86, 2014.

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

[3] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 205–214, Washington, DC, USA, 2010. IEEE Computer Society.

[4] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Oxford, UK, 1996.

[5] Leonardo Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1337–1342, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[6] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[7] Carl K. Chang. Changing face of software engineering. *IEEE Software*, 11(1):4–5, 1994.

[8] Kai H. Chang, James H. Cross II, W. Homer Carlisle, and Shih-Sung Liao. A performance evaluation of heuristics-based test case generation methods for software branch coverage. *International Journal of Software Engineering and Knowledge Engineering*, 6(4):585–608, 1996.

[9] J. Clarke, J.J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, June 2003.

[10] K. Deb and K. Deb. *Search Methodologies*, chapter Multi-objective optimization, pages 403–449. Springer US, 2 edition, 2014.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[12] Kalyanmoy Deb and Deb Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[13] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, July 1984.

[14] Juan J Durillo and Antonio J Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771, 2011.

[15] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, January 1996.

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[17] Gordon Fraser and Andrea Arcuri. The seed is strong: Seeding strategies in search-based software testing. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *ICST*, pages 121–130. IEEE Computer Society, 2012.

[18] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, February 2013.

[19] Matthew J. Gallagher and V. Lakshmi Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Trans. Software Eng.*, 23(8):473–484, 1997.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[21] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[22] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[23] Mark Harman. The current state and future of search based software engineering. In Lionel C. Briand and Alexander L. Wolf, editors, *FOSE*, pages 342–357, 2007.

[24] Mark Harman, Lin Hu, Robert M. Hierons, André Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1359–1366, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[25] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–12, 2015.

[26] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[27] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *ICST Workshops*, pages 182–191. IEEE Computer Society, 2010.

[28] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.

[29] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[30] B. Jones, H. Stahmer, and D. Eyres. Automatic tructural testing using genetic algorithm. *Software Enginnering Journal*, 1996.

[31] Fitsum M. Kifetew, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Paolo Tonella. Orthogonal exploration of the search space in evolutionary test case generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 257–267, New York, NY, USA, 2013. ACM.

[32] James C. King. A new approach to program testing. *SIGPLAN Not.*, 10(6):228–233, April 1975.

[33] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[34] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[35] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, August 1990.

[36] Kiran Lakhotia, Mark Harman, and Hamilton Gross. Austin: An open source tool for search based software testing of c programs. *Inf. Softw. Technol.*, 55(1):112–125, January 2013.

[37] Marco Laumanns, Lothar Thiele, Kalyanmoy Deb, and Eckart Zitzler. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary Computation*, 10(3):263–282, 2002.

[38] Christian Lücken, Benjamín Barán, and Carlos Brizuela. A survey on multi-objective evolutionary algorithms for many-objective problems. *Comput. Optim. Appl.*, 58(3):707–756, July 2014.

[39] Thomas J. Mccabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[40] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[41] Phil Mcminn. Iguana: Input generation using automated novel algorithms. a plug and play research tool. Technical report, 2007.

[42] Phil McMinn. Search-based software testing: Past, present and future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 153–163, Washington, DC, USA, 2011. IEEE Computer Society.

[43] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27(12):1085–1110, December 2001.

[44] Christoph C. Michael, Gary McGraw, Michael Schatz, and C. C. Walton. Genetic algorithms for dynamic test data generation. In *ASE*, pages 307–308, 1997.

[45] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Software Eng.*, 2(3):223–226, 1976.

[46] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Software Eng.*, 32(3):193–208, 2006.

[47] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *ICST*, pages 1–10. IEEE, 2015.

[48] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability*, 9:263–282, 1999.

[49] A. Rogers and A. Prugel-Bennett. Genetic drift in genetic algorithm selection schemes. *Trans. Evol. Comp*, 3(4):298–303, November 1999.

[50] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 1995.

[51] Simone Scalabrino. Progettazione e realizzazione di un tool per la generazione automatica di casi di test. Master's thesis, University of Salerno, 2015.

[52] Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an automated test generation tool to practice: from pex to fakes and code digger. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 385–396, 2014.

[53] N. Tracey. *A search-based automated test-data generation framework for safety critical software.* PhD thesis, University of York, 2000.

[54] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, ASE '98, pages 285–, Washington, DC, USA, 1998. IEEE Computer Society.

[55] N. Tracy, J. A. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *Proceeding of the IFIP International Workshop on Dependable Computing and Its Applications*, 1998.

[56] Alison Lachut Watkins. The automatic generation of test data using genetic algorithms. In *Proceedings of the 4th Software Quality Conference*, volume 2, pages 300–309, 1995.

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

[58] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation - GECCO 2004, Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004, Proceedings, Part II*, pages 1400–1412, 2004.

[59] L. Darrell Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 116–123, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[60] S Xanthakis, C Ellis, C Skourlas, A Le Gall, S Katsikas, and K Karapoulios. Application of genetic algorithms to software testing. In *5th International Conference on Software Engineering and its Applications*, pages 625–636, 1992.

[61] Yuan Yuan, Hua Xu, and Bo Wang. An improved nsga-iii procedure for evolutionary many-objective optimization. In Dirk V. Arnold, editor, *GECCO*, pages 661–668. ACM, 2014.

[62] Yuanyuan Zhang. The sbse repository. http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/.

[63] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search based requirements optimisation: Existing work and challenges. In Barbara Paech and Colette Rolland, editors, *REFSQ*, volume 5025 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2008.

[64] E. Zitzler, M. Laumanns, and L. Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical report, Technical Report 103, Swiss Federal Institute of Technology, 2001.

[65] Eckart Zitzler and Simon Künzli. Indicator-based selection in multiobjective search. In *PPSN*, volume 3242 of *Lecture Notes in Computer Science*, pages 832–842. Springer, 2004.

Acknowledgements

My model for business is the Beatles: They were four guys that kept each other's negative tendencies in check; they balance each other. And the total was greater than the sum of the parts.

Steve Jobs

At the end of my university studies, I would like to thank all those people who made this experience so unforgettable for me. I would never have been able to reach this goal without the guidance of my professors, the help and the support from friends and from my family.

Foremost, I would like to express my gratitude to my supervisor, Prof. Andrea De Lucia, for his guidance, his expertise and his accuracy. I could never have chosen a better supervisor for my thesis work. I would like to thank Dario, who patiently corrected my writing, for all stimulating discussions and continuous advices.

I am very thankful to my adventure companions of the last two, fantastic years, here in Fisciano. Working with you, was the most beautiful, formative and challenging experience of my life. I thank Simone, who collaborated to OCELOT: working together, I have learned to appreciate his character so brilliant and, all the way, so joker. I thank Matteo, who taught me the value of the self-confidence, for our conversations and for his incomparable advices; if I had an older brother, I would like that he looked like him. I thank Carlo

for teaching me the importance of commitment and determination, besides for all burgers and beers that we enjoyed together. I sincerely hope, wherever life will lead us, to still enjoy your company.

My sincere thanks also goes to all people who motivated and encouraged me: Francesca, my family, my parents and all my friends.

At last, I thank Mina, and I hope that she can be proud of me today.