Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Evolutionary Testing of Stateful Systems: a Holistic Approach

Doctoral Dissertation of:
Matteo Miraz

Advisor:
Prof. Luciano Baresi
Coadvisor:
Prof. Pier Luca Lanzi
Tutor:
Prof. Gianpaolo Cugola
Supervisor of the Doctoral Program:
Prof. Barbara Pernici

2010 – XXII

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32 I-20133 — Milano

A chi mi ha accompagnato,
sostenuto e incoraggiato.

Abstract

Testing should be one of the key activities of every software development process. However it requires up to half of the software development effort when it is properly done. One of the main problems is the generation of "smart" tests to probe the system, which is both difficult and time-consuming. The research community has been proposing several ways to automate the generation of these tests; among them, the search-based techniques recently achieved significant results.

This doctoral dissertation presents *TestFul*, our evolutionary testing approach for stateful systems; it is tailored to work on object-oriented systems. It uses a *holistic* approach to make the state of object evolve, to enable all the features the class provides, and to generate the shortest test with the utmost coverage for the class under test. We employ several complementary coverage criteria to drive the evolutionary search. We aim to generate tests with high fault detection effectiveness. To this end, we consider the system from complementary perspectives and we combine white-box analysis techniques with black-box ones. The evolutionary search is completed with a local one, and we establish a synergic cooperation between them. The evolutionary search concentrates on evolving the state of objects, while the local search detects the functionality not yet exercised, and directly targets them.

All the proposal were subject to an extensive empirical validation. We devised a benchmark composed of independent benchmarks for tests, public libraries, and third party studies. As comparison, we consider both search-based, symbolic, and traditional (i.e., manually generated by human being) approaches. The achieved results were encouraging: *TestFul* efficiently generate tests for complex classes and outperforms the other approaches. The proposals presented in this dissertation open new interesting research directions. On one side, one can continue refining the search strategy, by considering more advanced search techniques and by leveraging more advanced coverage criteria. On the other side, one can adapt the approach to work either at a coarse-grained level —and focus on the integration testing— or on other kind of stateful systems (e.g., components or services).

Riassunto

Il test dovrebbe essere una delle attività chiave di ogni processo di sviluppo software. Tuttavia, se ben fatto, esso richiede fino alla metà dello sforzo complessivo richiesto per lo sviluppo del sistema. Uno dei principali problemi è la generazione di test "intelligenti", in grado di sondare a fondo il sistema. La loro generazione è un'operazione difficile e dispendiosa. La comunità di ricerca ha proposto diversi metodi per automatizzare la generazione di questi test, tra i quali, le tecniche basate sulla ricerca hanno recentemente ottenuto risultati significativi.

Questa tesi di dottorato presenta *TestFul*, il nostro approccio per generare test per sistemi con stato mediante algoritmi evolutivi. In particolare, *TestFul* si focalizza su sistemi orientati agli oggetti, probabilmente la tipologia più diffusa di sistemi con stato. *TestFul* utilizza un approccio olistico, che incentiva l'evoluzione dello stato degli oggetti usati nei test in modo da attivare tutte le funzioni fornite dalla classe, e quindi produrre il miglior test per la classe considerata. Per guidare la ricerca, *TestFul* si avvale di diversi criteri complementari per giudicare la copertura. In particolare, si avvale sia di tecniche black-box, sia di analisi white-box, al fine di considerare il sistema da diverse prospettive e garantire un'elevata capacità dei test generati di trovare i potenziali errori. Il processo di ricerca è completato con una ricerca locale. La ricerca evolutiva si concentra sull'evoluzione dello stato degli oggetti, mentre la ricerca locale rileva la funzionalità non ancora esercitata, e si focalizza direttamente su quelle.

L'efficacia delle proposte fatte è stata studiata mediante un'ampia e sistematica validazione empirica. A tal proposito, è stato proposto un benchmark composto da progetti considerati da altri studi empirici e da librerie pubbliche. Come confronto, consideriamo sia i lavori che utilizzano tecniche di ricerca, sia quelli che adottano l'esecuzione simbolica, sia gli studi che analizzano test generati da esseri umani. I risultati ottenuti sono incoraggianti, e confermano la capacità di *TestFul* di gestire in maniera efficiente anche le classi più complesse, superando le prestazioni degli altri approcci. Le proposte presentate in questo lavoro di tesi aprono nuove interessanti direzioni di ricerca. Da un lato, si può continuare a perfezionare il processo di ricerca, esaminando tecniche di ricerca più avanzate e sfruttando criteri di copertura più completi. D'altra parte,

si può cercare di far scalare l'approccio sia considerando diversi livelli di granularità, concentrandosi sui test di integrazione, sia esaminando altre tipologie di sistemi di con stato (ad esempio, componenti o servizi).

Contents

1. Introduction **1**
1.1. Contributions 3
1.2. Outline 4

2. A Holistic Approach **7**
2.1. Evolutionary Algorithms 8
2.2. Test Representation 10
2.3. Global Search 12
2.3.1. *TestFul*: The Tool 13
2.4. Design of the Experiments 15
2.4.1. Threats to validity 19
2.5. A First Empirical Evaluation 20
2.5.1. Tuning 21
2.5.2. Experimental Results 23
2.5.3. Conclusions 30

3. Efficiency Enhancement Techniques **31**
3.1. Local Search 32
3.2. Seeding 37
3.2.1. Test Adaptation 39
3.3. Fitness Inheritance 41
3.4. Combined Improvement 44

4. Guidance **47**
4.1. Coverage of the Behavioral Model 47
4.1.1. Empirical Evaluation 52
4.2. Coverage of the Control-Flow Graph 55
4.2.1. Local Search 55
4.2.2. Empirical Evaluation 56
4.3. Coverage of the Data-Flow Graph 62
4.3.1. Local Search 63
4.3.2. Empirical Evaluation 67

5. Related Work **77**
5.1. Non Search-Based 77
5.1.1. Specification-Based 77
5.1.2. Symbolic Execution 78
5.2. Search-Based Approaches 79
5.2.1. Blind search 80
5.2.2. Guided search 82

6. Conclusions and Future Research Directions **87**

A. Mutation Testing **91**

Bibliography **92**

List of Figures

2.1. Representation of Tests. 11
2.2. *TestFul*-generated JUnit test. 12
2.3. Architecture of *TestFul*. 14
2.4. *TestFul* as an Eclipse plug-in. 15
2.5. Tuning the size of the object repository. 23
2.6. Simple State Machine: (a) behavioral coverage and (b) def-use pairs coverage achieved by random testing and *TestFul*. Curves are averages over ten run. 25
2.7. Hard State Machine: (a) behavioral coverage and (b) def-use pairs coverage achieved by random testing and *TestFul*. Curves are averages over ten run. 26
2.8. Coffee Machine: (a) behavioral coverage and (b) def-use pairs coverage achieved by random testing and *TestFul*. Curves are averages over ten run. 28
2.9. Fraction: (a) behavioral coverage and (b) def-use pairs coverage achieved by random testing and *TestFul*. Curves are averages over ten run. 29

3.1. Two loops of *TestFul*. 32
3.2. Hybridization. 36
3.3. Coverage on class `Fraction` with a random search. 37
3.4. Seeding. 38
3.5. Average performance improvement with class adaptation on package `java.util`. 41
3.6. Speed-up with Fitness Inheritance. 42
3.7. Fitness Inheritance. 43
3.8. Combining Seeding, “Best” Hybridization, and Uniform Fitness Inheritance. 46

4.1. ADABU’s object behavioral model for the `Vector` class. . 48
4.2. Behavioral model of class `BoundSet`. 50
4.3. Mutation Analysis on class `Fraction`. 54
4.4. Average structural coverage. 59
4.5. Branch Coverage vs. time limit. 61

5.1. Def-Use pairs coverage with random testing. Complete coverage consists of 100 def-use pairs. 81

List of Tables

2.1. Benchmark. 16

3.1. Improvement with hybridization. 36
3.2. Improvement with seeding. 38
3.3. Improvement with Fitness Inheritance. 43
3.4. Combined improvement of Seeding, "Best" Hybridization, and Uniform Fitness Inheritance. 46

4.1. Performance. 60
4.2. Average Statement Coverage. 72
4.3. Average Branch Coverage. 72
4.4. Average number of def-use pairs covered. 73
4.5. Average number of predicate-use covered. 73
4.6. Mutation score. 75
4.7. Mutation score comparison: p-value of the statistical hypothesis testing $H_0 : mean(A) > mean(B_i)$ vs. $H_1 : mean(A) \leq mean(B_i)$ with $B := \{$ "Carleton Code", "Carleton State", "Sannio Code", "Sannio State", "*TestFul* Basic" $\}$. 75

1. Introduction

Our life is more and more permeated by software systems, and their reliability is becoming critical to the modern society. The National Institute of Standards and Technology confirms this trend and estimates the yearly cost of software errors in 60 billion U.S. dollars [Tas02]. Although research in software validation and verification recently achieved several important results, the most common approach to assess the quality of software systems is through testing. However, testing might turn out to be extremely expensive requiring up to half of the software development effort [Bei90].

More in particular, testing a software system in a systematic and reliable way requires two distinct actions to take place: (i) the generation an optimal set of “smart” tests to excercise the software as an overall system and (ii) a rigorous way to compare the outputs of the system undergoing the testing to catch and report any abnormal or unexpected behavior.

Generating a “smart” and as small as possible set of tests is not a trivial task and might require a lot of effort. The research community has been working on proposals to alleviate the burden by generating tests (semi-)automatically. Among these, search-based approaches (see for example [MMS01]) were particularly interesting. They deal with functions with no internal state (e.g., libraries of functions) and rely on the *divide and conquer* paradigm to exercise each feature separately. In contrast, many modern systems (e.g., object-oriented systems) are *stateful*. The behavior of the different functions heavily depends on their internal state. This characteristic adds complexity to the testing problem [PY04], but it also allows to better exploit the internal states so to exercise the different features without re-starting from scratch every time, thus saving resources.

Tests that exercise single classes, isolated from the rest of the system, are called *unit tests*. However, even at this low level of granularity, it is not possible to exercise a class with all possible input values (even if we consider a function with only one integer parameter, there are 2^{32} possible configurations). Instead of chasing the “complete” test for a system, tests are judged with respect to their fault-detection ability. However, the measurement of tests' fault detection effectiveness is undecidable.

For this reason, tests' thoroughness are often judged with respect to some adequacy criterion. For example, the *branch adequacy criterion* requires that tests execute each branch of the class. This information can be used to measure the quality of tests by calculating the coverage of selected ones. For example, the *branch coverage* is defined as the ratio between the number of branches exercised by a test and the total number of branches of the system being tested. The higher the coverage is, the closer the tests are to fulfill the branch adequacy criterion. Albeit there are not proofs that more coverage will make a test suite more effective in finding faults, in general better coverage is associated with better tests. Additionally, coverage is easily measured, while test effectiveness is in contrast undecidable.

System undergoing a systematic testing process also require a rigorous way to catch and report unexpected behavior. In this respect, test oracles allow to verify if a method behaves correctly, comparing its result with the expected one [BY01]. Oftentimes, the developer is in charge of this task, manually verifying the correctness of the output of the system. Modern frameworks, such as jUnit [jun], allows one to embody those checks directly in the tests, and automates the verification. Even if currently the developers are in charge of writing those control statements, there are advanced techniques to perform these checks automatically. For example, it is possible to leverage the mechanism of contracts [Mey92] and automate the testing process [LCO+07]. Despite the effectiveness of this technique, it is rarely used in development [Xie05]. Accordingly, several alternative heuristics have been proposed that, for example, check for program crashes [MFS90, KKS98] or uncaught exception [CS04, CSX08] to pinpoint an unexpected behavior.

Our research focuses on the generation of "smart" tests for stateful systems more than in the definition of oracles. We adopt a search-based technique, and we leverage an evolutionary algorithm to generate the best[1] test for the system. Differently from other approaches that adopts a *divide and conquer* paradigm, we prefer a *holistic* approach. This makes the state of the objects evolve, and leverages the states reached to efficiently exercise all the behaviors of the system. The goal is to generate the test with the utmost quality, measured in terms of some given complementary coverage criteria. To detect errors, we reuse the research results achieved in this field, and we support both the mechanism of contracts [Mey92, LBR99, LCC+03] and the heuristics based on uncaught exceptions [CS04, CSX08].

[1] The best for the system is the one that *efficiently* achieves the chosen adequacy criteria.

1.1. Contributions

This dissertation presents a novel approach able to efficiently generate tests for stateful systems. The main contributions are:

Search-Based Software Testing Despite the seminal work of Tonella [Ton04], most of the research on search-based software testing still focuses on single functions, aiming to cover all their branches. Instead, object-oriented systems are widely diffused, and more complete coverage criteria have been proposed. Accordingly, this research directly targets this kind of systems, and considers several complementary coverage criteria.

Holistic Approach We adopt an alternative approach to generate tests for stateful systems —in particular for object-oriented ones— that exploits internal states to save on effort. The proposal models test generation as a search problem, which is addressed through an evolutionary algorithm. Since the internal state of stateful systems tightly relates the features exposed by the system, instead of chasing each feature separately, we prefer a *holistic*, incremental approach. The internal states reached with previous tests are then used as starting points for the next ones. Each test is rewarded according to the number of features it exercises, and implicitly this means that we reward the ability of a test to put the system in interesting states. We use this measure to drive the evolutionary engine and focus the effort of the search close to the best tests generated so far. By recombining tests already able to exercise different features (i.e., able to reach interesting states), it is likely that we can activate further new features.

Efficiency Enhancement Techniques The research around evolutionary algorithms proposed several efficiency enhancement techniques to improve the efficiency of the search process. We consider them to increase the performance of our approach, and faster converge to the optimal test for the considered class. Indeed, we deeply leverage one of them, the *local search*, and we propose a **hybrid approach**. We establish a synergistic cooperation between the (global) evolutionary search and the local search, and the results of one are used as the starting point of the other. The evolutionary search concentrates on evolving the state of objects, while the local search detects the functionality not yet exercised, and directly targets them.

Complementary Coverage Criteria Despite most of the approaches only consider the coverage of the control-flow graph —and they mainly focus on the utmost level of branch coverage— the ability of these coverage criteria to assess the quality of tests has been disputed [HFGO94]. Other coverage criteria only consider some aspects of the system, hence they might fail to detect important errors. Consequently, we use complementary coverage criteria to judge the quality of tests. In particular, we combine white box criteria, namely the *statement coverage*, *branch coverage*, and *all def-use pairs coverage*, with a novel black box criterion, which rewards tests according to their *behavioral coverage*. These coverage criteria analyze the system from different perspectives, hence their combined evaluation is able to better judge the quality of tests. Moreover, we propose a *pluggable architecture*, which easily allows one to extend the system and to introduce novel ways to judge the quality of tests.

Extensive Empirical Validation All the proposal presented in this dissertation are supported by an extensive empirical evaluation. To this end, we select several classes from independent benchmark for tests, public libraries, and third party studies. We also compare our approach against the state of the art in test generation, considering *search-based*, *symbolic*, and *traditional* (i.e., manually generated by human beings) approaches. We compare the performances in terms of the coverages of the control-flow graph and the data-flow graphs. To better measure the fault detection ability of the generated tests, we also use the mutation analysis. In order to achieve more trustable results, we run the simulation several times, and we calculate the average value and the standard deviation. Altogether, we run our tool for 4,748 hours of CPU-time (or almost 200 days of CPU-time), and we considered more than 200 classes.

1.2. Outline

The remainder part of this dissertation is organized as follows.

Chapter 2 introduces the holistic approach. This leverages evolutionary algorithms (briefly presented in the first part of the chapter) to generate tests. The approach is holistic, since it search for tests able to exercise all the features provided by the class by evolving the state of objects. The Chapter also presents the benchmark we use

to validate empirically the proposals, and performs a preliminary evaluation study.

Chapter 3 presents some efficiency enhancement techniques, which are able to significantly improve the performance of evolutionary algorithms. In particular, we focused on *local search*, *seeding*, and *fitness inheritance*. The chapter briefly presents them, explains how they are integrated in *TestFul*, and empirically evaluates their contribution.

Chapter 4 considers three complementary analysis techniques, aimed to evaluate the quality of tests from different perspectives. This information is used to provide the search strategies with enough guidance to fruitfully recombine tests and generate the "best" one (i.e., the one able to reveal most of the faults).

Chapter 5 analyzes the state of the art in automatic test generation, and considers both "traditional" approaches and search-based ones.

Chapter 6 concludes the dissertation, summarizing the main achievements and outlining the future research directions.

2. A Holistic Approach

Search-based techniques do not concentrate on solving problems, but scan the space of possible solutions to identify the “best” ones. Some of them adopt random search strategies: they pick elements randomly and keep the best ones. Others (e.g., hill climbing, evolutionary algorithms, simulated annealing) impose some guidance; they measure how close to the ideal solution each evaluated element is, and use this information to drive the exploration of the solution space.

The application of search-based techniques to software testing is not new [MMS01]. Search-based test generation approaches have already been used to reveal failures in widespread systems [CLOM07]. Instead of analyzing the implementation or the specification of the system to generate the tests, they generate (maybe randomly), evaluate, and refine directly the sequences of operations that represent the tests.

These solutions, however, have still some issues. The works that adopt a heuristic to drive the search process focus on a single element of the control flow graph (e.g., a particular branch) at a time. Consequently, these approaches (e.g., [Ton04]) need several runs to create a complete test suite for the class under test. Instead, we must consider that the tests for *stateful* systems —and Java classes are good representatives— are conceptually composed of two parts: the first creates the desired state of the system, while the second exercises the actual behavior. A smart test generation approach must be able to exploit some synergies and thus reuse the same state for testing different behaviors. The same first part can be shared among different seconds parts, without any need for recreating the same initial state repeatedly.

Moreover, targeting each element of the control flow graph separately could be misleading or provide insufficient guidance. For example, Ferguson [FK96] shows that a condition may depend on others, and this dependency may not be explicit in the control flow graph (but a combined analysis of the control and data flow graphs is required to highlight it). If we target these conditions separately from their dependencies, we waste effort and reduce the overall efficiency of the approach. For this reason, we want an approach able to use different kind of guidance, and judge tests with different and complementary analysis techniques.

These considerations about efficiency, guidance, and reuse are the un-

derpinnings of *TestFul*, our proposal for testing Java classes.

We leverage evolutionary algorithms (briefly presented in Section 2.1) to generate tests. Accordingly, we had to choose a proper representation for tests (Section 2.2) and how to discriminate between good elements and bad ones (Section 2.3, which applies our holistic approach and explains the global search). Section 2.5 validates the whole approach empirically.

2.1. Evolutionary Algorithms

Evolutionary algorithms are search methods inspired by the principles of natural selection and genetics. They maintain a population of candidate solutions that are evaluated using a *fitness* function. Operators inspired by natural selection focus the search on the most promising individuals (or candidate solutions), and operators inspired by genetics recombine and mutate parts of existing individuals to discover better candidate solutions. The schema of a typical evolutionary algorithm is reported as Algorithm 1. Initially, a population of individuals is randomly generated (line 1). Then, the following four steps are repeated (line 2) until a termination criterion is met. First, individuals in the population are evaluated by computing their *fitness* to estimate their capability of solving the problem (line 3). Next, selection is applied to generate the population P_s containing the individuals which should survive to the next generation (line 4). Recombination is applied to the individuals of P_s with probability p_χ (line 5) and then mutation is applied to the resulting population with probability p_μ (line 6). Recombination takes two individuals, acting as parents, and produces two offsprings by mixing their genetic material so that good features of the parents (good building blocks [Gol02]) may be reassembled to produce potentially better candidates. Mutation applies small random changes to the individuals in the

Algorithm 1 Evolutionary Algorithm.

1: $P \leftarrow$ RandomPopulation();
2: **while** Criterion Not Met **do**
3: $\quad P_f \leftarrow$ EvaluateFitness(P);
4: $\quad P_s \leftarrow$ Select(P_f);
5: $\quad P_\chi \leftarrow$ ApplyRecombination(P_s, p_χ);
6: $\quad P_\mu \leftarrow$ ApplyMutation(P_χ, p_μ);
7: $\quad P \leftarrow P_\mu$;
8: **end while**

population. Finally, the new population P_μ replaces the old one (line 7) and the cycle continues.

There are two key decisions involved in the design of an evolutionary algorithm: how to represent candidate solutions and how to evaluate them. Representation defines what type of genetic material the evolutionary algorithm will recombine and mutate. Typically, a candidate solution is represented as a chromosome consisting of a sequence of genes. Recombination splits two genes (two candidate solutions) and mixes the resulting parts. Mutation applies random modifications to the content of a gene. The evaluation of candidate solutions is encoded in the computation of the fitness function that measures the quality of individuals to ensure that better candidates will have more reproductive opportunities. Ideally, the fitness function should provide effective guidance towards the optimum solution.

Multi-objective evolutionary algorithms. The fitness function provides only one criterion to judge the quality of a candidate solution. However, in many applications there are more, sometimes competing, criteria to select the best solution. For instance, in this work we look for short test sequences that also cover as many cases as possible. Therefore, we try to both maximize the coverage and minimize the length of the test sequences. *Multi-objective evolutionary algorithms* tackle problems with multiple objectives and extend the basic framework (Algorithm 1) by (i) introducing more fitness functions and (ii) modifying the selection step (Algorithm 1, line 4) to take them into account.

In this way, the evolutionary algorithm targets the pareto frontier between various objectives. The result of the search process is the set of individuals that provide the best compromise among the different objectives, and looks for the shortest tests with the best coverage. A multi-objective evolutionary algorithm tries to reach the pareto frontier by working on the non-dominated set of solutions. Elements belonging to this set represent the best compromise found so far. The evolutionary algorithm uses recombination to push the non-dominated set as close to the pareto frontier as possible.

TestFul employs NSGA-II [DAPM02] as multi-objective evolutionary engine, which is able to evolve the population of tests towards the best solution according to the used coverage criteria. NSGA-II focuses on the non-dominated set: it is able to achieve good coverage on the class under test, and high efficiency at the same time.

Hybrid evolutionary algorithms. The individuals in a population inherit qualities from their parents, but during their lifetime they also try to improve themselves (e.g., by seeking better living conditions).

Inspired by this observation, some researchers introduced the concept of *hybrid evolutionary algorithms* that couple evolutionary algorithms and methods for local search. In this case, after selection and crossover have been applied, each individual has the opportunity to improve itself by applying (for a limited number of steps) a local search (e.g., a greedy search) to reach a nearby (sub-)optimum value. Differently from the real-world evolution, the local search is performed by modifying the genetic representation of individuals. The generated offspring inherit not only the parents' original chromosomes, but also the achieved improvements. Empirical studies showed that hybrid evolutionary algorithms can be faster and more efficient than "standard" evolutionary algorithms since they can exploit the local information available in the surrounding of good candidate solutions.

Parallelization. *TestFul* allows *master-slave* parallelization to speed-up the fitness evaluation by distributing the execution of tests across available processors. Note however that, *TestFul* works on a single population, only fitness evaluations are distributed onto available processors, while selection, crossover, and mutation are performed on the entire population.

2.2. Test Representation

Evolutionary algorithms require a careful selection of the way tests are represented. Good representations allow crossover and mutation operators to easily recombine good parts of the elements and form a high-quality offspring, thus the evolutionary algorithm converges faster to the optimal solution of the problem.

Even if there exist representations tailored to particular kind of classes (e.g., for containers [Arc10b]), *TestFul* uses a representation able to deal with any Java class. This is composed of two parts: some *variables* and a *sequence of operations* (see Figure 2.1).

The former are shared among all tests, and comprises variables for any class that might be involved in the test; all these classes form the *test cluster*. To calculate the test cluster, *TestFul* analyzes the class under test (CUT) and by transitively including the type of all parameters of all public methods (and constructors). Since abstract classes and interfaces can be used as formal parameters, *TestFul* asks the user for additional classes (i.e., concrete implementations of the abstract data types) and adds them to the test cluster.

To enable polymorphism, it stores an object of type `A` either in a variable with the same type or in a variable whose type is an ancestor of

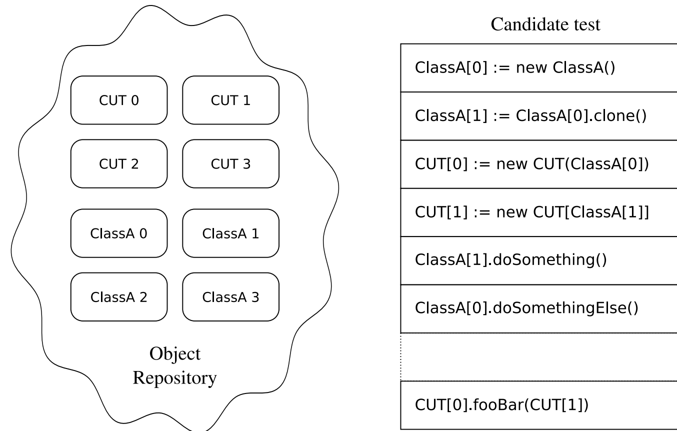


Figure 2.1.: Representation of Tests.

`A` (i.e., `A`'s super-classes or `A`'s implemented interfaces). Conversely, when an object is selected from a variable of type `A`, it may be an instance of `A` or of one of its subclasses.

Each test starts with all the variables set to null (or uninitialized in case of primitive types). The sequence of operations works on the aforementioned variables by using the values they contain or by storing new ones. In this way, a test can evolve the content of the variables, and let objects reach complex state configurations.

We consider the following operations:

- **assign** assigns a primitive value —i.e., boolean, byte, integer, long, float, double— to a variable in the context with the proper type.

- **create** creates an object by using available constructors, and store its reference in a variable of the context. As for parameters, it uses objects and primitive values taken from variables in the context. If one of the parameters has a primitive type and the used variable has not been initialized, the operation is not valid and it is skipped.

- **invoke** invokes a method. The receiving object and actual parameters are taken from variables in the context. If the method returns a non-void value, it may be stored in a variable of the context. Note that if the method mutates the state of some objects picked from the context, the change will be reflected on subsequent operations using the same variable. Like with **create**, if one of the used parameters is a primitive type and it has not been initialized, the operation is skipped since it is not valid.

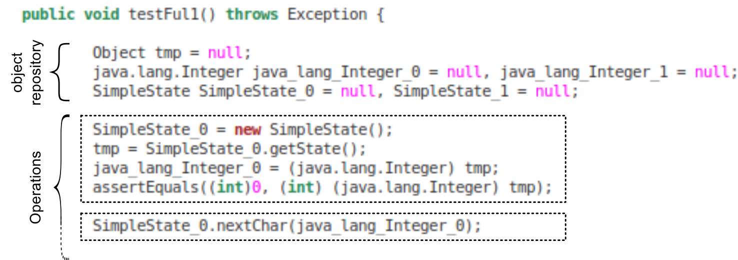


Figure 2.2.: *TestFul*-generated JUnit test.

TestFul allows the user to save the results of the test-generation process as jUnit tests. The conversion process first performs some transformations, so to simplify tests by pruning irrelevant operations and by splitting long tests into shorter ones, making them human-comprehensible. Afterward, each test is executed once again, and the behavior of the class being tested is monitored. This collected behavior acts as oracle for regression tests, and it is used to create assertions that verify whether newer versions of the class acts like the tested one. Finally, tests are converted and saved as jUnit tests (see Figure 2.2 for an example). The structure of each jUnit test is close to the one internally used by *TestFul*, exception made for the assertions. Each test starts with the declaration of the variables, that is the object repository. Afterward, there is the sequence of operations that constitutes the test. Each operation works on those variables, which are used as input parameters, as object accepting the method call, and to store the result of the invocation. Operations are followed by some assertions, to verify whether the behavior of the class is in line with the one registered during the test generation.

2.3. Global Search

Stateful systems are particularly tedious to test. One must put objects in proper states and provide the correct values for the input parameters of the functions under test. Other search-based approaches either do not use any guidance (e.g., random testing), or do not take in account the internal states of objects, even if they explicitly target stateful systems. Moreover, guided approaches apply the *divide and conquer* paradigm, and chase each feature exposed by the system separately from the others. In contrast, the features exposed by stateful systems are often tightly related, since they depend on the internal state. Reaching the proper configuration of the object's state can be expensive, but it could enable

different features, and new state configurations can be reached from it.

This is why we propose *TestFul*. It addresses Java classes and uses a holistic approach to generate tests for a class. It also enables the reuse of state configurations to exercise different features.

The fitness function in charge of driving the evolutionary algorithm exploits coverage information: the higher the coverage is, the better. *TestFul* monitors the execution of tests and calculates the coverage they achieve on the class under test. A good coverage likely means that the test is able to exercise significant parts of the class under test and to put objects in interesting states, which in turn ease the selection of further tests to exercise new parts of the class. *TestFul* employs a multi-objective evolutionary algorithm [ZLT01] to select tests according to their scores, recombine them, and finally it generates the test for the class under test.

The architecture of *TestFul* allows one to plug different kinds of analysis techniques, aimed to measure different types of coverage criteria (Chapter 4 presents three different coverage criteria, but one can easily make *TestFul* use other criteria). Hence, the fitness function *f(t)* of a test t when n criteria are used is:

$$f(t) = \langle t.length, cov_1(t), \ \dots, cov_n(t) \rangle$$

Since we want to ensure efficiency, the test's size ($t.length$) must always be minimized, while all coverage criteria $cov_i(t)$ must be maximized.

To generate new tests, *TestFul* takes two sequences from the previous generation, and cuts their list of operations at a random point. Children are obtained by recombining adequately the four pieces. The first (second) child is generated by concatenating the first part of the first (second) parent with the second part of the second (first) parent. Since the cut points may be different in parents, it is likely that one child becomes longer than the other. The recombination of variable-length individuals tends to produce tests that are long enough to adequately cover the class under test. However, the evolutionary algorithm is guided to penalize unnecessarily long sequences.

Mutation modifies new individuals to avoid local optima. We propose a simple mutation that may randomly (i) remove an operation from the test, or (ii) add a randomly generated operation at a random point of the test.

2.3.1. *TestFul*: The Tool

TestFul is fully implemented in Java, and it is available at `http://code.google.com/p/testful` as an open source project licensed under GPL v3.0 (`http://www.gnu.org/`).

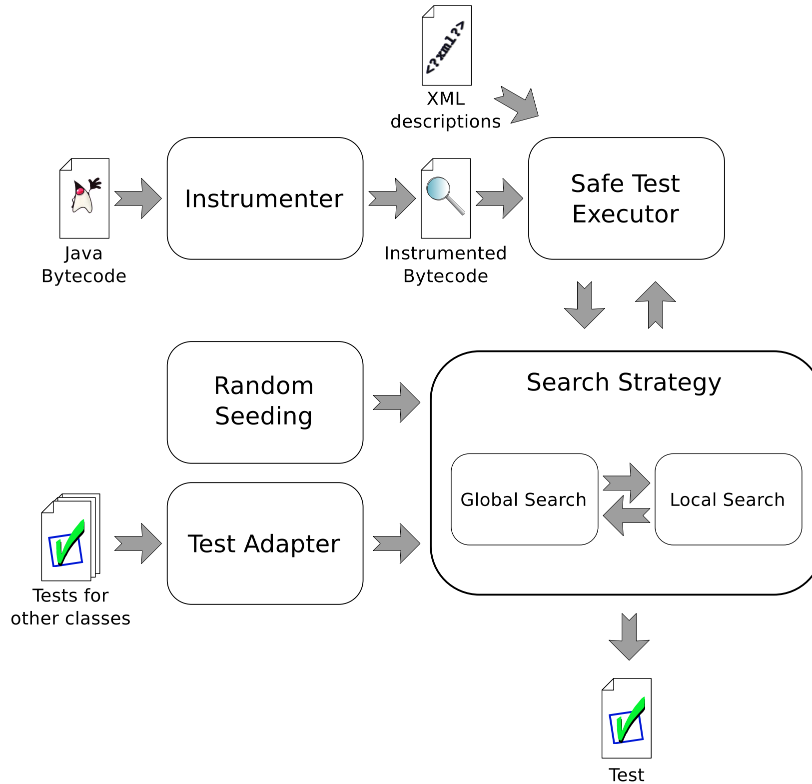


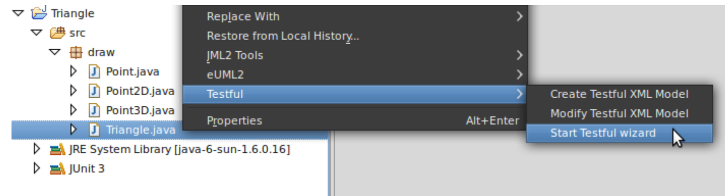
Figure 2.3.: Architecture of *TestFul*.

Figure 2.3 reports the architecture of *TestFul*. The *Search Strategy* is the main module of *TestFul*. It uses two kind of searches, namely the *global search* and the *local search*[1], to generate the test with the utmost coverage on the system. This module interacts with *Random Seeding* and *Test Adapter* modules. These implement the seeding efficiency enhancement technique and provides the *Search Strategy* with a better initial population (Section 3.2 explains this technique). The *Search Strategy* module relies on the *Safe Test Executor* to execute the candidate tests it generates, and calculate their level of coverage. The *Safe Test Executor*, as the name suggests, manages the execution of tests and ensures that crashes in the classes under test does not stops the test generation process. Moreover, this module provides the *master-slave* parallelization technique, as it allows one to distribute the test execution among a cluster of computers. The *Safe Test Executor* depends on the *Instrumenter* module to modify the bytecode of the classes under test and insert the

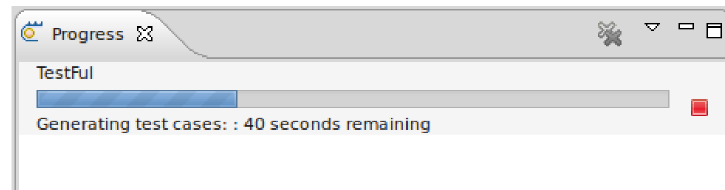
[1]The *local search* module will be introduced in Section 3.1.

tracking code required to measure the coverage of tests.

TestFul is released also as Eclipse plug-in, which integrates the Java Development Tool allowing one to easily create tests for his own classes. It is extremely simple to use: the user has only to right-click on the class to be tested (Figure 2.4(a)), provide some information (e.g., the behavioral abstractions), and wait until the generation process finishes (Figure 2.4(b)).



(a) Starting *TestFul*.



(b) *TestFul* in action.

Figure 2.4.: *TestFul* as an Eclipse plug-in.

2.4. Design of the Experiments

To evaluate the ability of *TestFul* to generate tests, we devised a benchmark composed of several project. All the empirical evaluation we present in this dissertation use (part of) this benchmark. For each considered project, Table 2.1 shows the number of lines of code without comments (LOC), the number of classes under test (#CUT), and the average cyclomatic complexity (CC).

This benchmark comprises projects taken from independent benchmarks for test-generation approaches and public libraries. Note that the projects contain some "easy" classes that act as *data-structure* and do not contain branches. We chose not do target these classes because also a naive tool would be able to reach their complete coverage in a couple of seconds.

Array Partition This code is part of the Quicksort algorithm, one of the fastest and most efficient sorting algorithms in use today. Quick-

Table 2.1.: Benchmark.

Project Name	LOC	#CUT	CC
Array Partition [DER05]	38	1	3.75
Binary Heap [DER05]	97	1	3.27
Binary Search Tree [DER05]	180	1	2.35
Commons Math v. 2.1 [Apa]	31,814	102	2.12
Disjoint Set [DER05]	130	2	3.37
java.util v. 1.6.0_20	9,668	15	2.33
JGraphT v. 0.8.1 [JGr]	10,394	88	2.10
NanoXML Lite v. 2.2.1	1,046	1	3.23
OrdSet [DER05]	231	1	3.14
Red Black Tree [DER05]	509	1	3.70
Roops [roo]	3,215	14	2.83
Siena [DER05]	1,099	4	3.05
Stack [DER05]	160	2	2.10
StateMachine [MS07]	75	2	10.00
Total	58,656	234	—

sort is extremely common, used explicitly in many programs and implicitly in others, since it is part of the Java libraries. Since it so frequently used, correctness is required, and proving this correctness for all situations is worthwhile.

This class divides an array into upper and lower halves, using the first element (a[0]) as a pivot. It does this by moving all elements whose value is larger than the pivot to after all the elements whose value is smaller than the pivot. Ideally, this pivot will be the median value, and after partitioning the array would be evenly divided between low and high values. It should also be noted that the algorithm uses an in-place partitioning scheme to save memory. As the name indicates, this class is implemented with an array rather than a linked list or tree of any kind.

Binary Heap The project contains an implementation of a binary heap, which is a data structure which has all the constraints of a binary tree and some additional ones to make it well formed.

Binary Search Tree The project contains an implementation of a binary search tree, which is a data structure that have low memory overhead and still allow fast (less than $O(N)$ in all but the worst case, $O(log(N))$ if the tree is balanced) searches.

Commons Math The Apache Commons Maths is a lightweight, open source, self-contained library that addresses mathematics and statistics problems, and it is widely used in several products.

In particular, this project contains the class `Fraction`. This is an immutable class that represents fractions and provides the basic operations between them. Since `Fraction` is immutable, it can be considered a particular instantiation of the object-oriented paradigm. This characteristic hinders the approach behind *TestFul*, since it is not possible to evolve the state of created objects. Nevertheless, we chose to include this kind of class in our benchmark and verify the behavior of *TestFul* on immutable classes, since these classes are widely used in real-world systems.

Disjoint Set This project contains two implementations of the *Disjoint Set* structure. This handles dynamic partitions of a set X composed of the first n integers, where n is specified when the object is instantiated. Each implementation ensures that the union of all subsets is X, and the intersection between any two subsets is empty. At the beginning there are n subsets, one for each element in X. The user can thus `join` two subsets, or `find` the subset a number belongs to. The authors [DER05] provide two implementations of this class: a one marked as *reference*, and a one marked as *fast*, which is more challenging.

package java.util The Sun's `java.util` package contains several implementations of the `Collection` interface. A collection in Java represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not, some are ordered and others unordered. This package is commonly used to evaluate automated test-generation approaches.

JGraphT JGraphT is an open-source library that supports graph transformations. It supports different algorithms, and supports various types of graphs including directed and undirected graphs, graphs with weighted / unweighted / labeled or any user-defined edges, various edge multiplicity options, including: simple-graphs, multigraphs, pseudographs, unmodifiable graphs, and subgraphs graphs that are auto-updating subgraph views on other graphs

NanoXML Lite The NanoXML project is a small open source XML parser for Java. It provides several implementations, and we use the *Lite* one, which provides a full XML parser in a single class. The extensible markup language, XML, is a way to mark up text in

a structured document. It is designed to improve the functionality of the web by providing more flexible and adaptable information identification.

OrdSet The OrdSet project contains a single Java class, which represents a bounded, ordered set of integers. The OrdSet class provides methods for adding a single element, removing a single element, and creating the union of two ordered sets. Despite its small size, it has a complex internal data structure. Consequently, the statechart that models its behavior and its control-flow graphs are comparable with those describing bigger projects.

Red-Black Tree This project contains only a single class, whose internal data structure is the *Red-Black tree*, a self-balancing tree with `search`, `insert`, and `remove` methods with a time complexity of $O(log(n))$. For this reason, the same data structure is used in Sun's implementation of the class `java.utils.TreeMap`, which is used in several related work as benchmark.

Roops This project was used as benchmark for a competition between bug finding tools and automatic test case generators for Java and C#. It defines a common notation for writing benchmarks for tools that decide Reachability in Object Oriented ProgramS (*Roops*). Roops is general enough to be useful for many different researchers, working on different tools for different applications and different languages. A common benchmark notation with standardized reachability goals should allow the community to share and build upon each others benchmarks, and enable researchers to compare empirical results.

Siena Siena (Scalable Internet Event Notification Architecture) is an Internet-scale event notification middleware for distributed event-based applications deployed over wide-area networks, responsible for selecting notifications that are of interest to clients (as expressed in client subscriptions) and then delivering those notifications to the clients via access points.

The `Java` version of Siena[2] is logically divided into a set of six components (consisting of nine classes of about 1.5KLOC), which constitute a set of external components C, and a set of 17 other classes of about 2KLOC, which constitute an application that could be constructed using C.

[2] Siena is available both for `Java` and for `C++`.

Sorting This project contains classes able to manage a sequence of numbers, allowing the user to add a number at the end of the list and to retrieve the sorted list. The class implements the following sorting algorithms: *Insertion Sort*, *Shell Sort*, *Heap Sort*, *Merge Sort*, and *Quick Sort*.

Stack This project contains two implements the stack data structure with LIFO (Last-In, First-Out) access policy. The two implementations differ on the internal data structure: one uses an array, while the other adopts a list. Both of them check for overflows (the user tries to insert an element in a full stack) and underflows (the user tries to remove an element from an empty stack).

State Machine This project is inspired by a function present in the VIM text editor [vim]. It implements a state machine with ten states that verifies whether the user activates a certain functionality by providing a particular sequence of ten characters.

The code was previously analyzed in [MS07], where the authors showed that a long random sequence of invocations on a single object can easily traverse the state machine towards the accepting state.

We extended the previous state machine and created a variant, called `HardStateMachine`, in which if a wrong character is supplied, the object is put in an erroneous state and all subsequent characters are discarded. In this case, the generation of meaningful tests is much more difficult since it has to generate the exact sequences of invocations that traverse the state machine, *without wrong characters*.

Vending Machine This project contains class `CoffeMachine`, a variant of the well-known `CoinBox` [KGH+95]. It simulates a coffee machine in which users can select the desired drink by invoking method `chooseTarget`; they can insert coins, and as soon as the price of the selected beverage is reached, they can get the desired drink by using method `get()`.

2.4.1. Threats to validity

This section analyzes the main threats that can affect the validity of the experiments we performed [WRH+00].

Threats to conclusion validity concern issues that affect the ability to draw the correct conclusions between the "treatment" and the outcome

of the experiments. To limit these threats, we did not interfere anyhow with the experiments, and ran the experiments on dedicated servers. Moreover, to limit the random fluctuations of performance, we selected a benchmark composed of several projects, and we performed several runs for each class.

Threats to internal validity concern factors that might affect the results of the experiments but are not controlled in the experiments themselves. To limit these threats, we ran *TestFul* using its default setting and we asked for the advice of experts on how to use symbolic execution. Moreover, our solution is based on results published in flagship conferences and journals.

Threats to construct validity concern the way we judged the outcome of the experiments. To this end, we performed two different kinds of experiments. The first one judges the quality of tests according to structural and data-flow coverages. It is widely accepted that the higher the coverage is, the better tests are, even if a high coverage is not enough to guarantee high quality. For this purpose, we also performed a second experiment using mutation analysis, which confirmed the results of the first experiment and limited these threats [ABLN06].

Threats to external validity concern external factors that might limit the generalization of the results. Although the results we show are limited to the considered projects, we carefully selected different types of software artifacts from different sources. In particular, our benchmark includes the `java.util` package, which has been widely used also in other empirical studies, and two widely used software libraries, namely `JGraphT` and Apache's `Commons Math`.

2.5. A First Empirical Evaluation

This Section aims to validate the holistic approach behind *TestFul*. In particular, we want to analyze the ability of the coverage criteria —used as fitness function— to drive the evolutionary algorithm towards the optimal test. In the positive case, if we analyze the coverage criteria during the evolutionary search, we would see them progressively increase (following a negative exponential), and eventually reach the optimal value. Otherwise, if the fitness function is not able to provide a good guidance, the coverage criteria would have a ragged evolution, composed of steps mixed with long plateaus.

Moreover, we also validate the efficiency of our proposal by comparing it against a random search approach. In particular, we use a Java implementation of *AutoTest* [MCLL07] as a yardstick, since it is one of the

most advanced proposal using a random search. Despite its simplicity, random search proved to be effective and it outperformed more advanced techniques [Ham06, CPL+08, PLB08, Arc10b]. In fact, it allocates all the computational effort in the exploration. In contrast, *TestFul* tends to be more incremental, and the effort is divided between an exploitation phase and the exploration phase (implemented respectively through the crossover and the mutation operators).

Both approaches are asked to maximize two complementary coverage criteria: the *behavioral coverage* and the *all def-use pairs coverage*.

The former (deeply explained in Section 4.1) monitor the test execution to infer a behavioral model of the class being tested. The completeness of the inferred model only depends on the quality of the executed test, hence we use this information as criterion to judge tests.

The latter requires to analyze the data-flow interactions of the Class Under Test enabled during the execution of the test. When a statement assigns a value to a variable, it is said to **define** the value of the variable. Similarly, a **use** is a statement that uses the value of a variable. A **definition-use pair** (*def-use pair*, or *du-pair*) is a couple of statements in which the former sets the value of a variable and the latter uses it. The **all def-use pairs** coverage criterion measures the quality of generated tests according to their ability to exercise all these pairs [RW85].

As a benchmark, we used four classes: a class implementing a simple state machine inspired to a piece of code of the VIM text editor [vim], a more challenging version of the same state machine, a variant of the well-known `CoinBox` [KGH+95], and a class taken from the Apache Commons Maths library [Apa].

All these classes are free of errors: our analysis does not consider the issue of detecting failures, but instead it focuses on measuring the ability of the approaches to generate tests that reach high coverage criteria. The detection of failures and the identification of errors is outside the scope of this work. The tool supports the mechanism of contracts [Mey92] and — if contracts are missing— employs the simple heuristic used in [PLEB07].

For each class, we applied both random testing and *TestFul* ten times and evaluated the quality of the obtained results in terms of (i) def-use pairs coverage, (ii) behavioral coverage, and (iii) size of the result. In this way, we evaluate the capability of a test to reach high coverages, its generation cost, and its suitability for regression testing.

2.5.1. Tuning

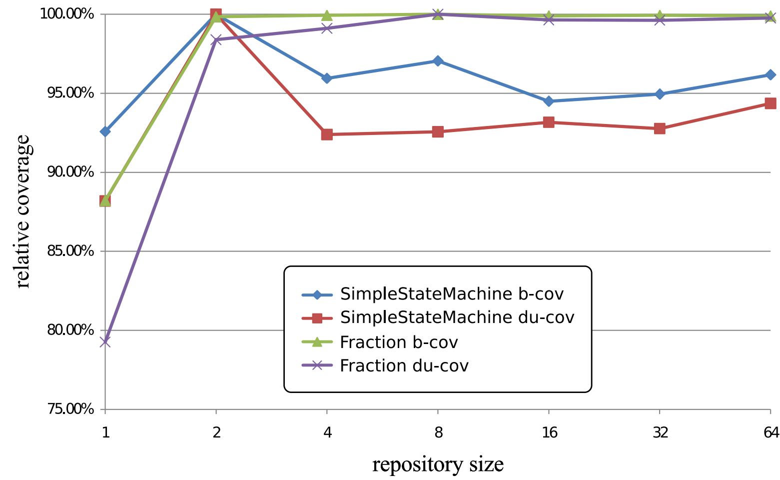
To compare random testing and *TestFul* we first needed to determine the best configuration for each method. In particular, we focused on two

aspects: the generation of random elements and the set up of the object repository.

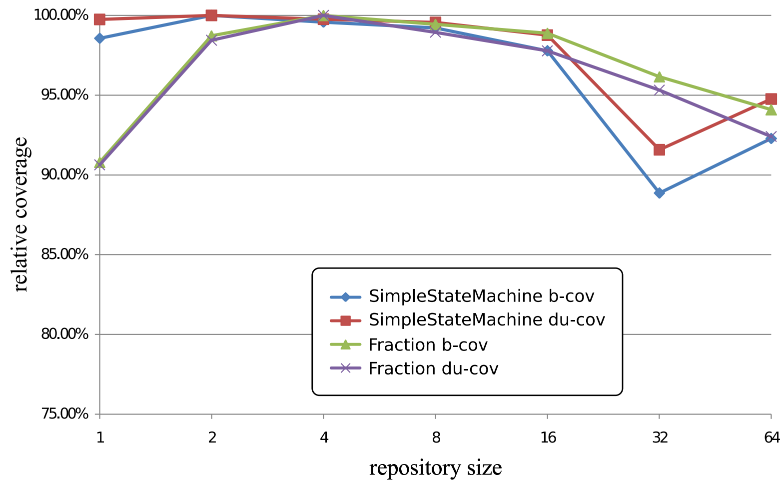
Generation of Random Elements. Both random testing and *TestFul* need to create random operations. This step represents the core of random testing, whose tests are nothing more than a long random sequence of operations. In *TestFul*, this operation generates the initial random population and it is also used to mutate tests. Although random element generation may exploit domain-specific information, in this study, we followed the approach by Ciupa et al. [CLOM07] and implemented plain random generation, without any use of domain-knowledge. In this way, the differences between the performance of random testing and *TestFul* only depend on the exploration strategies.

Object Repository. Both random testing and *TestFul* generate tests that work on a repository of N objects. The size of the repository is an obvious crucial factor for performance. On the one hand, small repositories may prevent the creation of a good test case. For example, if we consider class `Fraction`, a repository containing one object per class would prevent any mathematical operation with two or more fractions. On the other hand, huge repositories mean that each object has a very small probability to be selected and very long sequences of invocations might be needed to achieve even simple results.

Interestingly, although the size of the repository is crucial for the performance of the generation processes, to the best of our knowledge, it has never been subject of an empirical evaluation. In contrast, we performed a set of experiments to determine the best repository's size for the two methods. First, we selected two of the four classes (namely, `Fraction` and the `SimpleStateMachine`) and applied random testing and *TestFul* on both of them with varying repository's sizes. For each configuration, we performed ten runs. The average behavioral coverage and the average def-use pairs coverage on the two classes is reported in Figure 2.5(a) and 2.5(b), respectively, for random testing and *TestFul*. The results confirm what anticipated: runs with either a small repository or a too large one have bad performance. These results witness that a good repository's size for random testing is between 2 and 8 (corresponding to the best configurations for `SimpleStateMachine` and `Fraction`, respectively). *TestFul* is less sensible to the size of the repository, and achieves better results on both classes when the repository hosts 4 objects per class. For this reason, in the experiments presented here, we run random testing and *TestFul*, respectively, with a repository of 8 and 4 objects.



(a) Random testing



(b) *TestFul*

Figure 2.5.: Tuning the size of the object repository.

2.5.2. Experimental Results

To have a fair comparison between the two methods, we set a limit on the computation time to generate a test. Each run lasted six hours of CPU time for a total of 480 hours (or 20 full days). We used an IBM System p5, with a Power5 1.5 Ghz CPU and a heap of 1 GByte. All the reported statistics are averages of ten runs. The comparison presented here takes into account (1) the behavioral coverage, (2) the def-use pairs coverage, and (3) the size of the result. In particular, the size is fundamental

for regression testing: if the approach is able to generate a reasonably compact test, would be convenient to use the same test to ensure the correctness of new versions of the class.

Simple State Machine. To analyze the perfomance of random testing and *TestFul* on this class, we used a simplified version of the random character generator used in [CLOM07] that,

- 40% of the times generates a special character: { '\0', ' ', '\n', '\r', '\t' };
- 40% of the times, generates one of the 89 keyboard characters, including alfa-numeric ones and punctuation.
- 20% of the times generates a number between 0 and 255, and selects the character with the corresponding extended ASCII code.

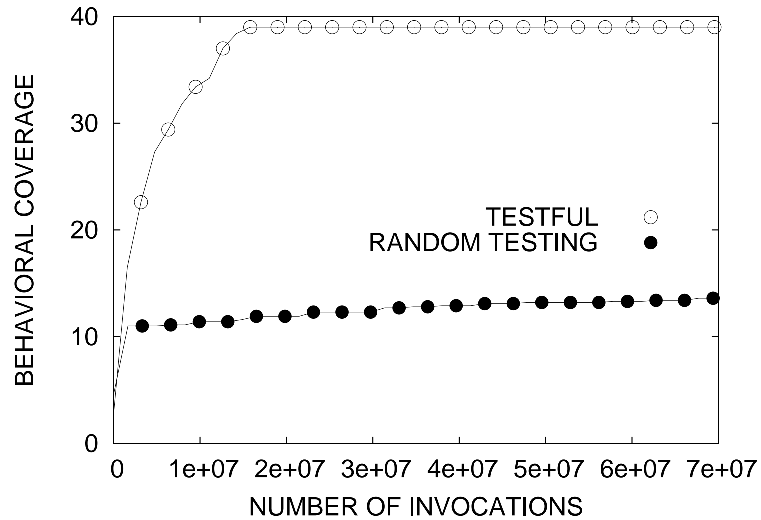
Each character accepted by the state machine has the same probability to be generated: $\frac{40\%}{89} + \frac{20\%}{256} \sim 0.528\%$.

Using this policy to generate characters, we run random testing and we noted that generated tests reach at most the second state of the state machine. The analysis of the runs showed that this was due to the high probability (25%) of replacing an object in the repository with a new one, thus vanishing the effort spent to reach the state. Accordingly, we reduced the probability of generating a new object to 1% to favor random testing on this sequential task.

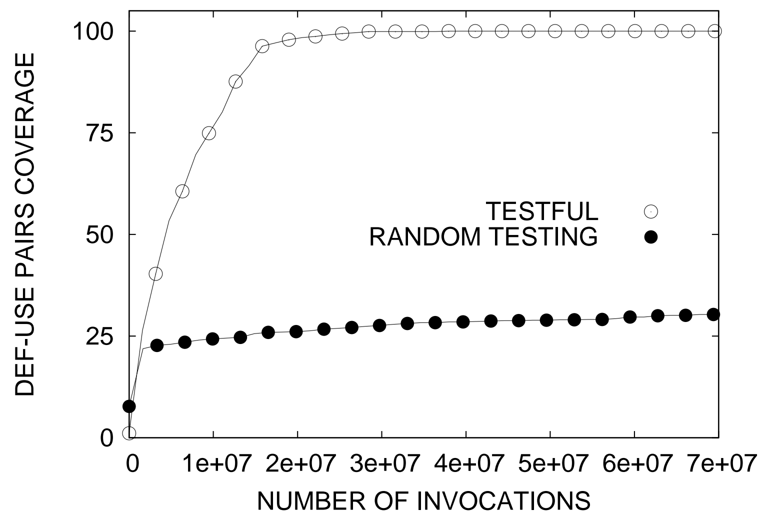
The behavioral and def-use pairs coverage for random testing (solid dots) and *TestFul* (empty dots) are reported in Figure 2.6. In this case, good behavioral coverage and def-use pairs coverage can only be achieved by identifying a sequence that activates all the possible states — a goal that is difficult to achieve with simple random search. In fact, in this case *TestFul* clearly outperforms random testing, generating a complete test in about one hour. Most interestingly, *TestFul* generates compact tests, with less than 500 operations.

Hard state machine. This class is more challenging than the previous one because any error in the input sequence brings the state machine into a halting (or absorption) state that vanishes all the effort spent to reach the previous state. Figure 2.7 reports the obtained performance. As in the previous case, our approach clearly reaches a better behavioral and def-use pairs coverage. Again, *TestFul* produces also compact test sequences, consisting of approximatively 40 operations.

It is worth noticing that neither random test nor *TestFul* were able to stress the class completely. The analysis of the runs shows that these results are caused by the error state: only one of the ten possible sequences

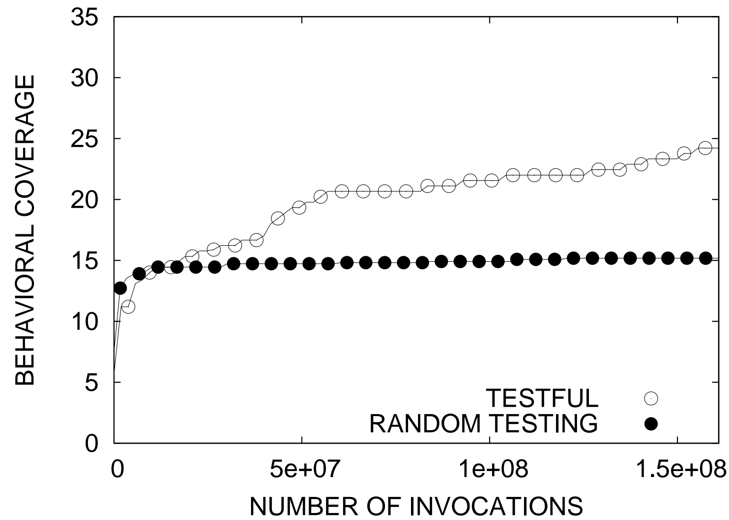


(a) Behavioral coverage

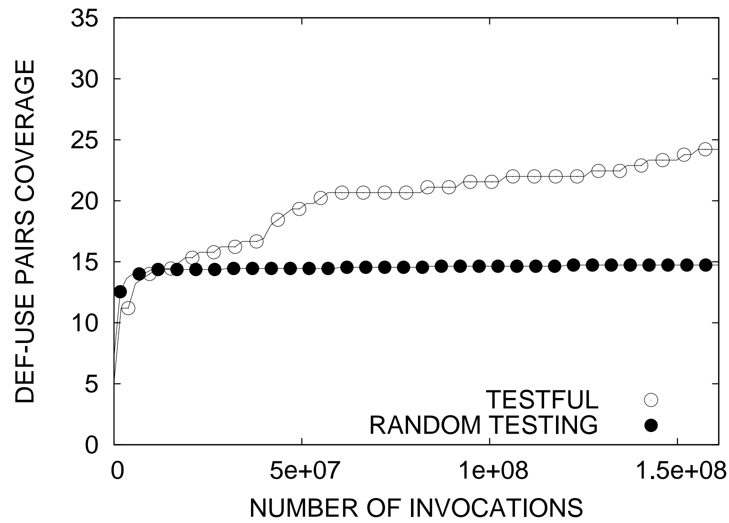


(b) Def-Use Pairs coverage

Figure 2.6.: Simple State Machine: (a) behavioral coverage and (b) def-use pairs coverage achieved by random testing and *TestFul*. Curves are averages over ten run.



(a) Behavioral coverage



(b) Def-Use Pairs coverage

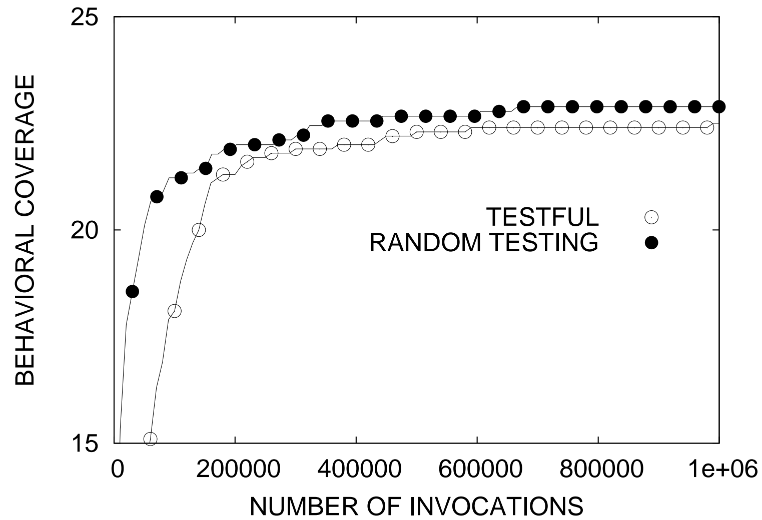
Figure 2.7.: Hard State Machine: (a) behavioral coverage and (b) def-use pairs coverage achieved by random testing and *TestFul*. Curves are averages over ten run.

reaches the accepting state. Note however that, although *TestFul* does not reach complete coverage within the time limit of six hours, it gradually explores the program's space and (at the end) its performance shows a slowly increasing trend. In contrast, the performance of random testing is clearly stucked at a level corresponding to the easily reachable states.

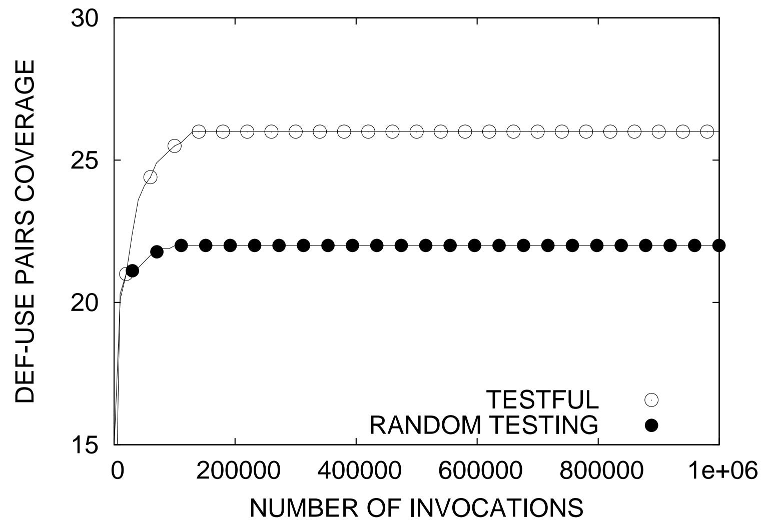
Coffee machine. On this class, both random testing and *TestFul* reached a complete coverage in a short amount of time (less than 10 minutes of CPU time). Accordingly, our analysis focuses on the performance achieved within the first million of operations. Figure 2.8 shows the results. Both approaches achieve a similar behavioral coverage while *TestFul* covered more def-use pairs. Moreover, as it happened in all the other cases, *TestFul* generated more compact tests (composed of around 50 operations), while random test required a much longer sequence of operations (around 300,000).

Fraction. Figure 2.9 presents the results for this class. Random testing reaches a better behavioral coverage than *TestFul* and it is also faster in obtaining the highest def-use pairs coverage. These results are easily explained by considering the characteristics of the problem. The class is immutable and therefore a series of invocations on the same object does not improve the test coverage. Only the creation of new objects in an undiscovered state leads to an improved coverage of the test. However, finding the correct values for calling the constructor is similar to finding a needle in a haystack. Without domain-specific knowledge, random testing scores better because it creates new instances of the fraction class more often, while *TestFul* tends to focus on the existing ones. As a consequence, random testing can achieve better behavioral and def-use pairs coverage. Note however that, although the class is simple and suitable for the approach, still random testing was not able to converge to an optimal result within the six hours of simulation, while *TestFul* achieved its sub-optimal result in approximately three hours. This fact suggests that *TestFul* has a better exploration strategy, and it will be able to reach even better results as soon as we use domain-specific knowledge.

An important aspect of automatic test generation is the size of the final test sequence: the shorter, the better. In this perspective, *TestFul* appears very promising since, although it did not achieved the same behavioral coverage as random test (for class `Fraction`), it generates test sequences that are dramatically shorter. In fact, *TestFul* generated test sequences consisting of 5,000 invocations on average. To reach the same coverage, random testing requires approximatively 200,000 invocations (i.e., 40 times more invocations than *TestFul*).

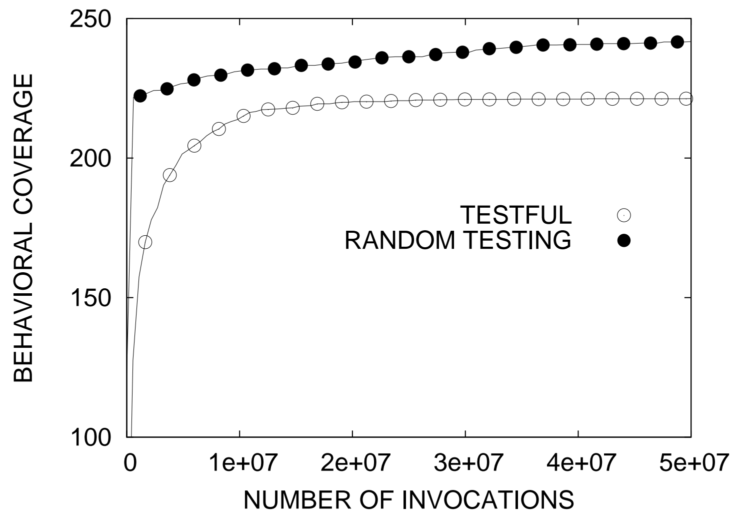


(a) Behavioral coverage

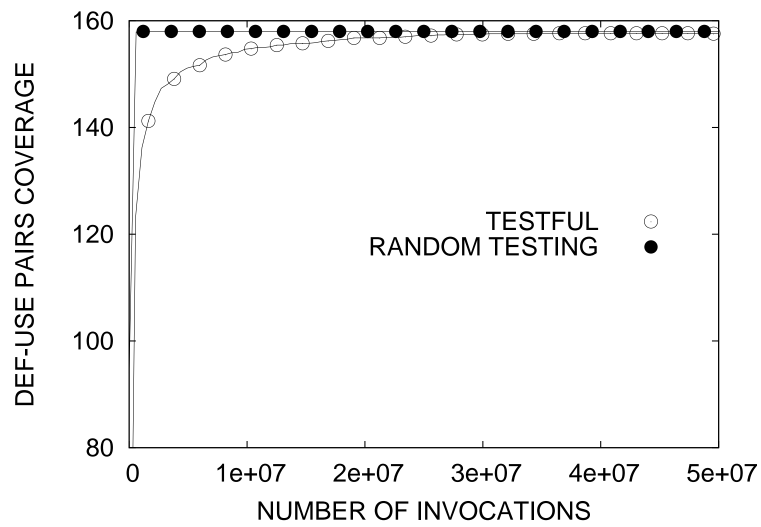


(b) Def-Use Pairs coverage

Figure 2.8.: Coffee Machine: (a) behavioral coverage and (b) def-use pairs coverage achieved by random testing and *TestFul*. Curves are averages over ten run.



(a) Behavioral coverage



(b) Def-Use Pairs coverage

Figure 2.9.: Fraction: (a) behavioral coverage and (b) def-use pairs coverage achieved by random testing and *TestFul*. Curves are averages over ten run.

2.5.3. Conclusions

Our results, although limited to four subjects, highlight some interesting aspects regarding the perfomance of random testing and *TestFul*. Firstly, in all the cases, *TestFul* generated the most compact tests. Secondly, random testing appears to explore the search space more widely, and achieves better results on extremely simple features. However, it performs poorly on classes with a complex state since its capability of enabling complex behaviors of the system under test appears to be limited. *TestFul* explores the search space less than random testing and therefore its performance decreases on classes with a simple (immutable) state, but with a huge number of possible combinations. However, *TestFul* can put objects in complex states and therefore it typically outperforms random testing on classes with complex states. Finally, both random testing and *TestFul* were not able to cope with particularly complex classes (e.g., the hard state machine). Our analysis of the runs witness that, in the case of random testing, this is caused by a lack of guidance; in contrast, for *TestFul*, this is due to a too coarse guidance.

3. Efficiency Enhancement Techniques

Chapter 2 presented the holistic approach behind *TestFul*, and we proved its ability to generate good tests for Java classes with complex internal states. However, *TestFul* is still computationally expensive and it requires a significant amount of CPU time [MLB09]. This is why this Chapter focuses on improving the evolutionary engine of *TestFul* through the use of modern efficiency enhancement techniques (EETs) [Gol02, Sas02, Sas07], and considers three classes of techniques: *seeding*, *local search*, and *fitness inheritance*.

This chapter explains each efficiency enhancement technique and evaluates their impact on the performance of *TestFul*. To this end, we devised a benchmark consisting of the six classes: `Disjoint Set Fast`, `Fraction` (from the *Apache Commons Math* project), `Red-Black Tree`, `Sorting`, `Stack Array`, and `State Machine` (see Section 2.4 for a detailed explanation). We compared the original version of *TestFul* against (i) the three enhanced versions using each of the enhancements and (ii) the version with all the enhancements altogether.

We run each version of *TestFul* ten times for 10 minutes of CPU time[1] each and traced the evolution of the best solutions found in terms of branch coverage. The higher the branch coverage is, the more valuable generated tests are. To perform the comparison, we first computed the average performance of the best individual during the evolution using the original and an enhanced version of *TestFul*. Figures 3.2, 3.4, 3.7, and 3.8 reports the performance plots for some of the experiments: x represents the time elapsed, while y summarizes the average branch coverage of the best test created so far. We calculated the average evolution speed as the integral of the average branch coverage with respect to time. We used this value to compare the enhanced version of *TestFul* against the base one, and measure the improvement ratio (Tables 3.1, 3.2, 3.3, and 3.4).

[1] We used a dedicated simulation server with an Intel Core2 Quad CPU Q6600@2.40GHz with 4 gigabytes of RAM

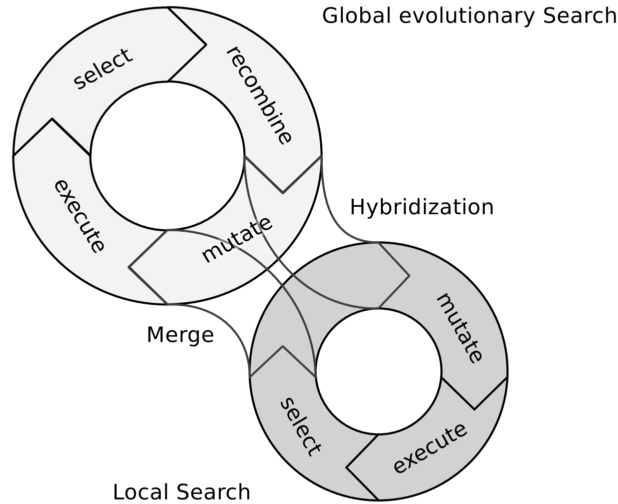


Figure 3.1.: Two loops of *TestFul*.

3.1. Local Search

The holistic approach behind *TestFul* rewards tests according to their ability to put objects in interesting states. To this end, Chapter 2 shows how the information contained in coverage criteria can be used to drive an evolutionary engine towards the creation of the optimal test for a class.

However, some classes have features that can only be tested via particular sequences of invocations with precise values as parameters. If the population of the evolutionary algorithm does not contain all the invocations needed, recombination and mutation can hardly generate a suitable test, quickly and reliably. Indeed, the fitness function that drives the global search does not provide enough guidance towards uncovered elements. For this reason, we hybridize the *global search* performed by the evolutionary algorithm by introducing a *local search* step that integrates the genetic operators. We establish a feedback loop between the two search: the results of one are used as starting point of the other. Consequently, this efficiency enhancement technique is deeply integrated in *TestFul*, whose resulting structure is shown in Figure 3.1.

The *global search* (depicted in light gray) prepares the state of the objects involved in the test using methods provided by the classes. We employ an evolutionary algorithm to search for the test that is able to reach all the interesting internal states. As fitness function, we use the level of coverage, according to some criteria (see Chapter 4) that the

test achieves: the higher it is, the more likely the test is able to reach interesting states.

The *local search* (depicted in dark gray) considers the tests evolved in the global search, analyzes their coverage criteria, and detects what is missing. Then, it adopts a search technique to modify the test and reach what was missing. If the search is successful, the modified test —which has a higher coverage of (at least) one criterion— is merged back in the population of tests evolved by the global search.

The local search only considers a small portion of individuals. In particular, we consider three policies: *best*, *5%*, and *10%*. The first one only uses the *best* test of the population; the other ones randomly select the 5% or 10% of individuals of the entire population.

On the selected tests, *TestFul* considers the coverage criteria to create a list of potential targets. A target is something that has not been covered yet, and its fulfillment would improve the test and raise at least one coverage criterion. Since we want "local" targets, we only consider those targets that can be reached by flipping the outcome of a single conditional statement. The search is thus "local" since it focuses on the neighborhood of a test and tries to improve it with respect to a single condition.

For each target, *TestFul* stores the number of attempts, and each time it focuses on the targets with fewer attempts. When the local search fails to reach a target, *TestFul* increases its number of attempts, and thus the next local search may prefer other targets. In several cases —for example, when a flag variable is used— covering other targets may also ease covering the current one. In other cases, a target simply requires more effort: this is why *TestFul* adjusts the maximum number of iterations of the local search according to the number of attempts done.

Among the targets with the same number of attempts, *TestFul* picks the easiest to reach. For this purpose, it analyzes the values used in the controlled condition. If one of them is a parameter of the method, it is possible to modify its value directly, thus the easiness of the target increases. Moreover, some conditions provide more guidance than others, thus targets controlled by the former are easier to reach than those controlled by the latter. To approximate the level of guidance, we consider the types of elements being compared. If the comparison is among numbers, it is possible to measure the distance to the desired outcome; conversely, the comparison of boolean values or references does not provide any clue.

Before starting the local search, *TestFul* simplifies the sequence of operations by pruning all the irrelevant ones. The resulting test is equivalent to the original one, since it is able to put objects in the same states

and exercises the same conditions in the same way.

As for the local search, *TestFul* employs a simple hill climbing. This creates a mutated version of the test, and verifies if its execution is closer to reach the target than the original version. If it is the case, the mutated version replaces the original one; otherwise the mutated version is discarded. The search continues until the target is hit or the maximum number of iterations is reached.

Even if hill climbing is a simple search technique, we experienced good results. This can be motivated by considering the particular context in which it is used: it has only to change the outcome of an exercised condition. Other works instead prepare the state of objects, reach the condition, and exercise the desired branch [Ton04]. This is why they adopt more powerful search techniques, such as evolutionary algorithms or particle swarm optimization.

To create a mutated version of the test, randomly we pick an operation and either we

- *remove* it. It may happen that removing an operation from the test facilitates the execution of the targeted branch. For example, if we wanted to check whether a collection is empty, we may remove insertion operations to shrink the collection and move it closer to the empty one. The test may also contain useless operations; removing them easies the search algorithm.

- *add* a random operation before or after it. The test may require some additional invocations to change the values of some conditions. For example, consider the check for inserting an element in a size-bounded collection: adding more elements to it helps augment the collection's size, eventually stressing the impossibility to add further elements.

- *change the values it uses*: if the selected operation stores a primitive value in a variable of the context (i.e., it is an *assign* operation), we try to change the stored value. We added a random value to the original one, or flip the value if it is a boolean variable.

In order to drive the search process, we use a simplified version of [WBS01]. Consequently, we focus on the condition that contains the targeted branch. That condition is in the form $a \oplus b$, where a and b are constants, local variables, or fields variables, and $\oplus$ an admissible relational operator. Since our instrumentation is performed directly on the bytecode, these are the only conditions we find. Those that involve more parts are translated at compile time in a set of simple ones. We

then monitor the execution of the test, and we focus on the condition that contains the targeted branch. Each time the condition is exercised, we record the values used in the evaluation, measuring the distance[2] as:

$$distance(a \oplus b) := \begin{cases} +\infty \leftrightarrow unsatisfied\ surroundings \\ -\infty \leftrightarrow target\ executed \\ |a - b| \leftrightarrow otherwise \end{cases}$$

This distance (its minimum value, if the condition is executed multiple times) is used as fitness function for the search algorithm. Note that, if the search process has to decide between two versions of the test with the same distance, we always prefer the shortest one. In this way the search converges faster, and the desired branch is reached earlier.

Sometimes the conditions that allow the local search to reach the target (*surroundings* in the formula) can be unsatisfied. This happens, for example, when the monitored condition is never executed. In this case, the local search pick a wrong direction and it moves away from the target; consequently *TestFul* gives the distance the maximum value.

Performance Evaluation. The data in Table 3.1 suggest that the performance improvements due to hybridization heavily depend on the type of the class under test.

Hybridization works well on classes with complex internal states that are difficult to reach, as in the case of the *State Machine*. Instead, when applied to a class with simple state and a huge number of configurations (e.g., class *Fraction*), hybridization may even slow down the convergence to the optimum (Figure 3.2(a)).

This can be explained by considering that, in this particular type of problems, the exploration of feasible configurations is more important than reaching a particular state. In the experiments discussed here, all versions of *TestFul* run for the same amount of CPU time. Accordingly, hybridization introduces an overhead that results in a decrease of the overall performance since it reduces the CPU time available for the exploration.

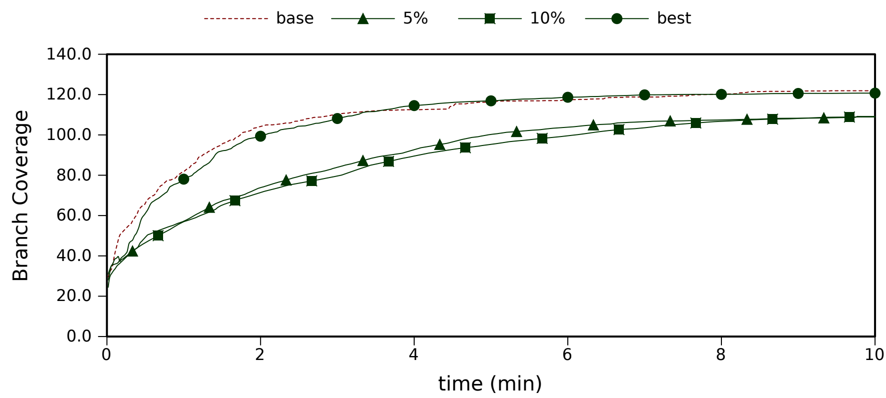
When applied to typical classes (with a moderately difficult internal state), the hybridization can lead to an interesting gain in performance, as shown in Figure 3.2(b) for class *Disjoint Set Fast*.

Overall, the results of Table 3.1 suggest that it is better to apply the local search on the best elements of the population (laying on the

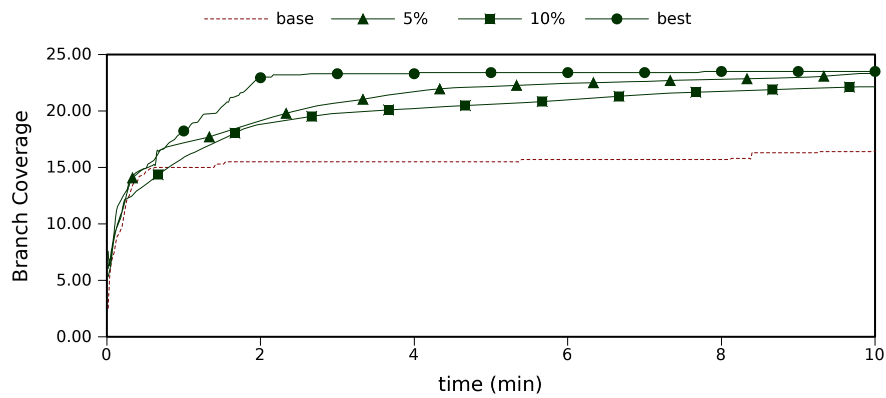
[2]For references, strings, and boolean values, we do not calculate the distance between a and b. Instead, if the condition is executed, but the branch is not reached, we set the fitness to a constant positive value.

Table 3.1.: Improvement with hybridization.

Class	Hybridization		
	best	**5%**	**10%**
Disjoint Set Fast	+43.94%	+35.61%	+28.13%
Fraction	-1.05%	-16.69%	-18.97%
Red-Black Tree	+64.78%	+24.01%	+6.07%
Sorting	+15.90%	+12.23%	+9.69%
Stack Array	+18.21%	+18.36%	+18.33%
State Machine	+399.97%	+347.49%	+315.39%
average	+90.29%	+70.17%	+59.77%



(a) Performance on class Fraction



(b) Performance on class Class Disjoint Set Fast

Figure 3.2.: Hybridization.

frontier). In fact, our novel hybridization technique that focuses on the *best* elements outperforms the traditional hybridization approach, which adopts a uniform sampling of the whole population.

3.2. Seeding

In [MLB09, MBL10], *TestFul* considered an initial population of randomly generated tests. However, such random populations usually have poor quality and contain tests that are not able to exercise any feature of the class being tested. As a consequence, *TestFul*'s evolutionary engine would start from a scarce supply of building blocks, which might hinder the search [Gol02].

To mitigate this problem, this work adds an initial seeding step to *TestFul* to improve the quality of the initial population and speed up the early stage of the evolutionary process (by providing better building blocks right from the start). For this purpose, we applied a very short step of random testing to generate the initial population of *TestFul*. Random testing simply performs random sequences of invocations and we only keep the best ones as tests. In our previous experiments [MLB09], we showed that random test usually finds good tests in the very first moments of its application, as Figure 3.3 shows. Accordingly, we seeded the initial population by applying a short 60-second run of random testing. This resulted in a higher-quality initial population and in an initial supply of good building blocks.

Performance Evaluation. Seeding leads to significant improvements in all the classes (ranging from 11.75% to a stunning 98.27%), but the

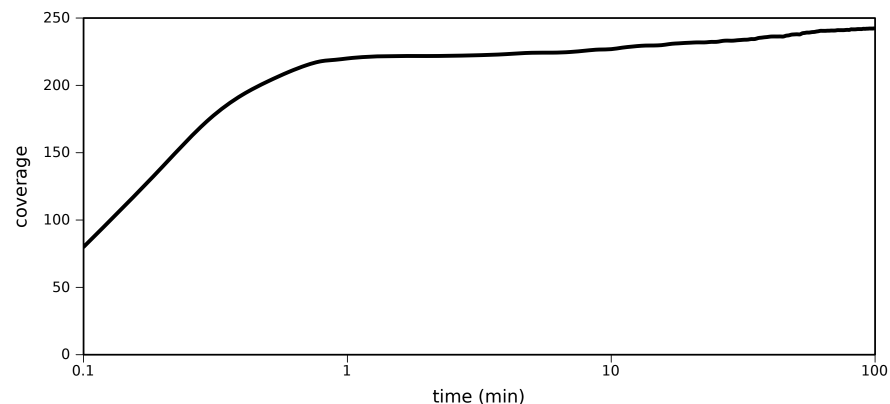
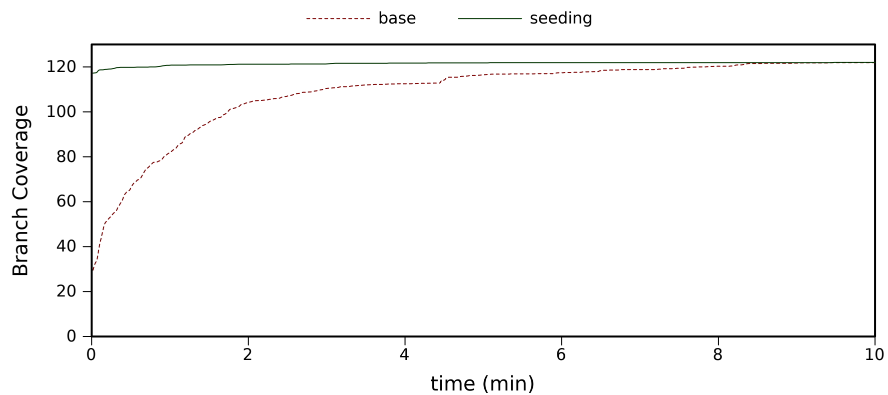


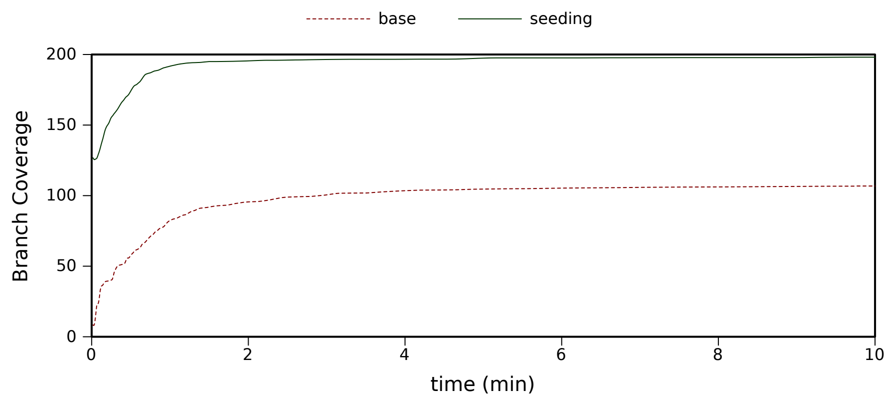
Figure 3.3.: Coverage on class `Fraction` with a random search.

Table 3.2.: Improvement with seeding.

Class	Seeding
Disjoint Set Fast	+53.79%
Fraction	+11.75%
Red-Black Tree	+98.27%
Sorting	+13.35%
Stack Array	+19.06%
State Machine	*same*
average	+32.70%



(a) Performance on class Fraction



(b) Performance on class Red-Black Tree

Figure 3.4.: Seeding.

State Machine. The initial population generated through random testing supplies good building blocks to the evolutionary engine that fruitfully mixes them and generates better tests quickly.

In simple problems, like the *Fraction*, seeding can almost solve the problem (Figure 3.4(a)): the branch coverage in the initial population is very close to the optimum. Class *Fraction* is however very simple and therefore also the original *TestFul* quickly reaches the optimum, and the overall improvement due to seeding is only 11.75% (Table 3.2). In [MBL10], we noticed that random testing was able to work better than *TestFul* on class *Fraction*. We explained this phenomenon by considering that the *Fraction* is a class with a simple internal state (just composed of a numerator and a denominator), but with a huge number of possible configurations. *TestFul* uses an incremental approach and works longer on discovered states. Instead, random testing does not impose any guidance and explores the search space wider: it performs better in early phases, slowing down later on.

The seeding technique exploits the initial good performance of random testing to create an initial population with good building blocks, so the evolution starts with a good branch coverage, very close to the optimal one. Hence, *TestFul* easily recombines the tests and generates optimal tests for the class faster than the normal version.

In more complex problems, like *Red-Black Tree*, seeding provides a better starting point for the evolutionary search and results in a faster convergence to the optimal solution (Figure 3.4(b)). In this case, the original *TestFul* cannot reach the same performance (in the limited time we set), accordingly, the improvement due to seeding is high, i.e., 98.27%.

Seeding is not effective on the *State Machine* (Table 3.2), where we have already shown [MBL10] that random testing cannot generate good tests even when applied for large amounts of CPU time. Accordingly, it is also unable to create interesting individuals for the initial population in *TestFul*.

3.2.1. Test Adaptation

The *test adaptation* mechanism performs a step further, and allows one to seed the initial population by *adapting* tests already created for other classes.

Often there are multiple implementations of the same feature, each tailored to particular needs. If one wants to test one of these implementations, it seems reasonable to use the tests created for the other implementations as starting point, and then focus on the particular requirement addressed by the considered class. In order to recognize if

two classes provide a different implementation of a same feature, *TestFul* verifies if they are in the same inheritance tree or if they implement a common interface. If it is the case, it is possible to reuse the tests generated for one class to seed the initial population of the other class. In this case, *TestFul* adapts the available tests, keeping all the method invocation that are supported by the class being considered. The resulting tests are then executed and ranked according to their coverage level. This is done because *TestFul*'s evolutionary algorithm uses a relatively small population size (512 individuals), hence it should keep only the most promising tests.

A particular case of this scenario is when one wants to test a new version of a class. Obviously, in this case the two versions of the class provides some features in common, and it is worth to use as starting point the tests generated for the previous version of the class.

Performance Evaluation. In order to evaluate the improvement given by the test adaptation, we used as benchmark the package `java.util`. This package provides several implementations of *containers* (i.e., `Collections` in the Java jargon), each to fit particular requirements and using a different internal data structure.

We treated the package as a single project, with no initial test available. Then, we run *TestFul* with the *test adaptation* feature enabled on all the classes of the package in sequence. The first class in the sequence did not reuse any test, the second class reused the tests generated for the first class, and the last class reused the tests generated for all the other classes in the package. In this way, we simulate a typical application of this feature when one wants to create tests for a project never considered before. Consequently, we judge the approach by considering the average performance of the package, calculated as the mean performance over all the classes.

As yardstick we run the version of *TestFul* with only the seeding technique based on a random-search. Indeed, it is known [Arc10b] that random search have very good performance on containers, hence we expect that the base version is particularly difficult to overcome.

Figure 3.5 reports the average performance improvement on package `java.util` when the test adaptation feature is enabled. As it is possible to see, it is worth to adapt tests created for other classes. *TestFul* is provided with better genetic material, and its performances are improved. This improvement is noticeable in both the early and the later stages of the evolution. Consequently, this technique is able to overcome a random search to supply an initial population of tests, and it also enable *TestFul* to generate tests with an higher level of coverage.

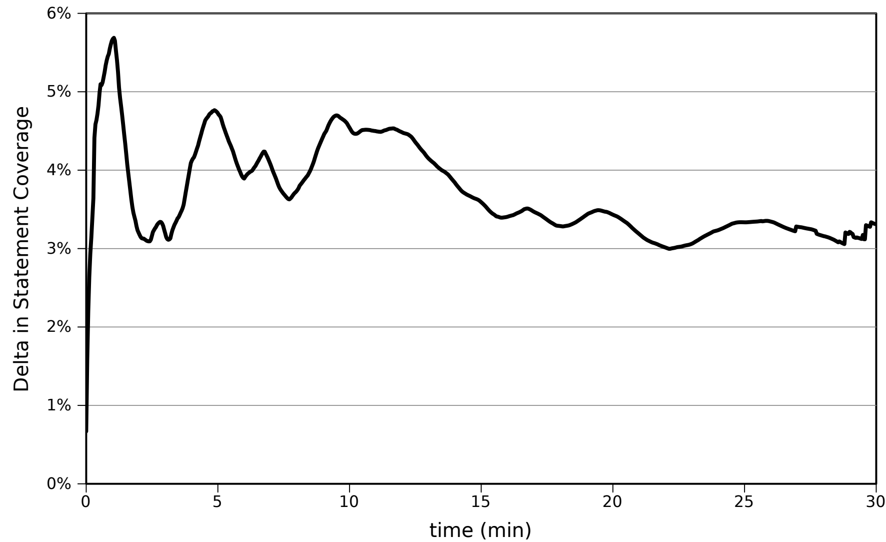


Figure 3.5.: Average performance improvement with class adaptation on package `java.util`.

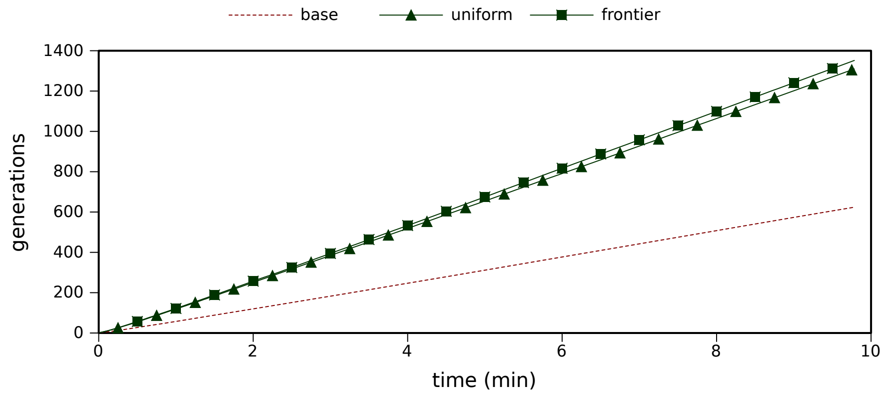
3.3. Fitness Inheritance

The evaluation of the fitness function is extremely expensive since it requires the execution of an entire test sequence, and thus test generation can be dramatically slow. To speed up the evaluation, we introduced the concept of *fitness inheritance*. At each generation, the usual fitness evaluation is only performed on part of the population, while remaining individuals (corresponding to the proportion p_i of the population) *inherit* the fitness from their parents. Fitness inheritance has been widely used and proved effective in many applications (e.g., [Sas02, Sas02, Sas07]).

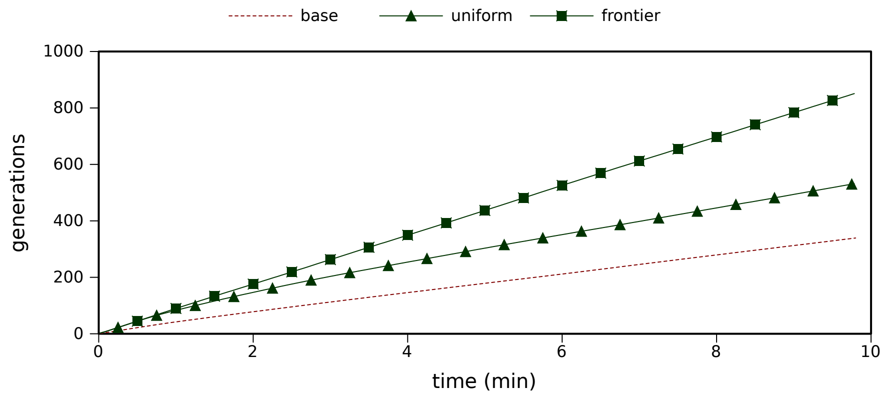
To add fitness inheritance in *TestFul*, we enhanced the crossover operator to calculate the inherited fitness of a newly created test as a function of its parents' fitness and their number of statements. Note that, such an inherited fitness is cheap to compute, but it only provides an approximation.

In *TestFul*, we tested two strategies to distribute the computational effort among individuals: *uniform selection* chooses the individuals randomly from the population by using a uniform distribution; *frontier selection* focuses on the best individuals on the frontier. The latter mainly evaluates the most promising individuals, making them less likely to inherit the fitness from their parents.

Performance Evaluation. Inheritance reduces the number of fitness evaluations, but it usually introduces noise that may hinder the convergence to the optimum. Hence, to analyze the effect of fitness inheritance in *TestFul* we have to consider two important aspects. On one hand, we must study the speed-up perceived by the evolutionary engine: each generation requires fewer evaluations, thus it is possible to process more generations within the same time-limit. For this purpose, Figures 3.6(a) and 3.6(b) show the average number of generations processed against the elapsed simulation time. As foreseen, fitness inheritance allows the evolutionary engine to complete more generations. Note that in complex problems (e.g., class Red-Black Tree), the gain is moderate with the *uniform* selection policy, while it is higher if the evaluation effort is focused



(a) Speed-up on class Stack Array

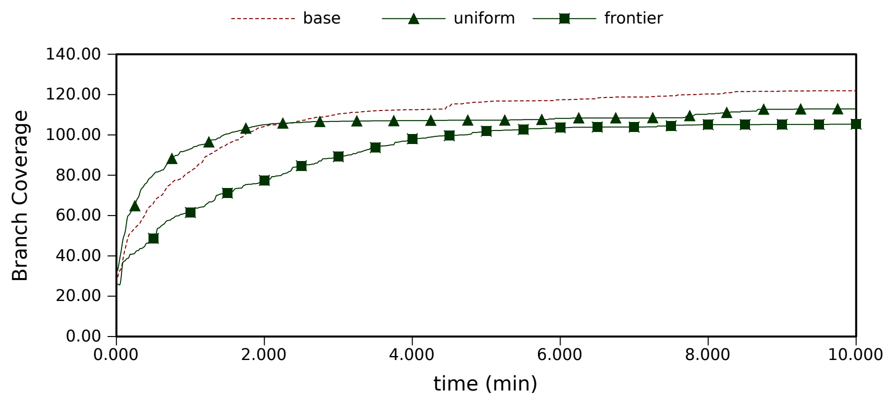


(b) Speed-up on class Red-Black Tree

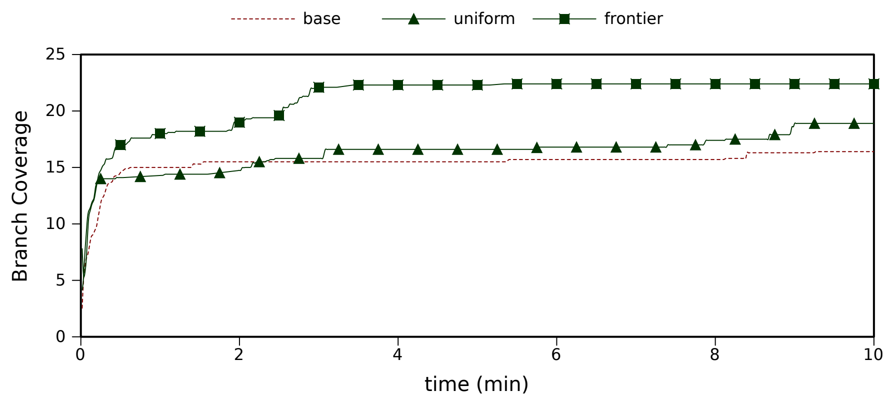
Figure 3.6.: Speed-up with Fitness Inheritance.

Table 3.3.: Improvement with Fitness Inheritance.

Class	Fitness inheritance	
	frontier	uniform
Disjoint Set Fast	+36.45%	+5.84%
Fraction	-15.72%	-3.98%
Red-Black Tree	-18.57%	+26.88%
Sorting	+13.58%	+16.50%
Stack Array	+0.55%	+1.73%
State Machine	*same*	*same*
average	+2.72%	+7.83%



(a) Performance on class Fraction



(b) Performance on class Disjoint Set Fast

Figure 3.7.: Fitness Inheritance.

on the frontier. The former might over-estimate long tests, leading to a population with longer elements. This phenomenon is limited if the elements on the frontier are evaluated more often.

On the other hand, we must analyze the ability of the evolutionary engine to deal with a noisy fitness function. To recognize whether the noise hinder the evolutionary process, we consider the average branch coverage achieved by the best individual. The improvement on all considered classes is positive, albeit limited. However, it heavily depends on the characteristic of the class being tested. For the *State Machine*, fitness inheritance results in the same performance. This is due to the poor guidance that the fitness function provides for this class; thus, reducing the number of evaluations does not help achieve better results.

In classes with a great number of simple states (e.g., *Fraction*), the recombination is more likely to generate tests that exercise new behaviors. However, this can only be detected when the test is evaluated, and fitness inheritance is likely to hinder the convergence (see Figure 3.7(a)). This phenomenon is amplified if we focus the evaluation effort on the frontier: the evolutionary engine focuses more on the same set of tests, ignoring other behaviors detectable by individuals not belonging to the frontier. Classes like *Disjoint Set Fast* (see Figure 3.7(b)) let the internal state evolve through an ordered sequence of method invocations. Consequently, the ability of a test to exercise certain behaviors heavily depends on the qualities of its parents, and the fitness inherited is very close to its actual value. Moreover, new behaviors are easier to reach by working on tests able to exercise more behaviors: better results are achieved when the evaluation effort if focused on the frontier.

3.4. Combined Improvement

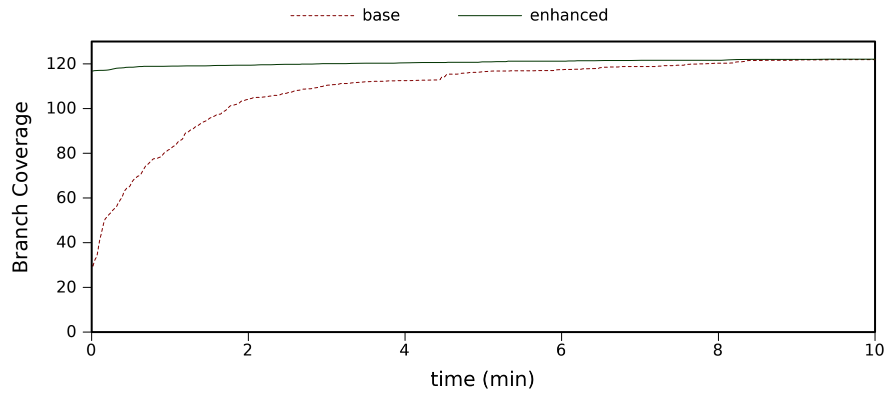
At the end, we enabled all the three efficiency enhancement techniques and chose the *best* hybridization policy and the uniform fitness inheritance. The results of Table 3.4 show that in all the classes there is a significant improvement over the original version of *TestFul*. In some cases, these efficiency enhancement techniques allow *TestFul* to generate faster a test with the same structural coverage (see the results for class *Fraction* in Figure 3.8(a)). However, they also allow *TestFul* to generate tests with higher quality (see the results for class *Sorting* in Figure 3.8(b)).

Note that different efficiency enhancement techniques can cooperate to increase the performance even more. This is the case of the *State Machine*: *hybridization* improves performance of 400%, while *fitness in-*

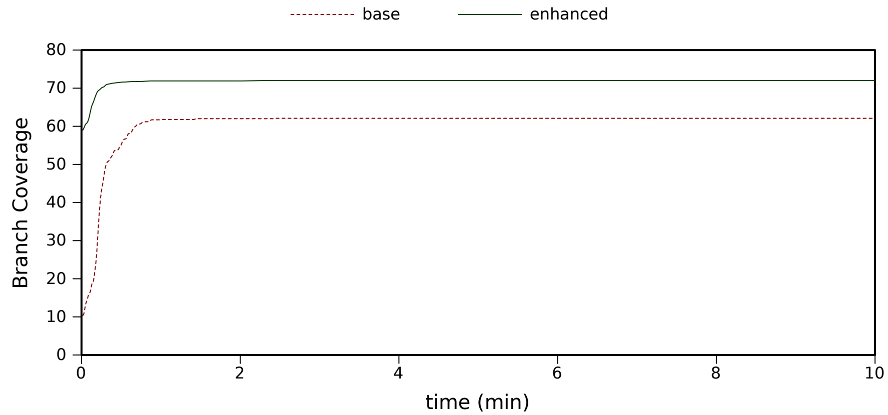
heritance has no effect. However, when applied together, *hybridization* enables *fitness inheritance* to contribute more in increasing performance, reaching a speed-up of 473%.

Table 3.4.: Combined improvement of Seeding, "Best" Hybridization, and Uniform Fitness Inheritance.

Class	overall
Disjoint Set Fast	+54.85%
Fraction	+10.96%
Red-Black Tree	+90.12%
Sorting	+18.28%
Stack Array	+19.06%
State Machine	+472.60%
average	+110.98%



(a) Performance on class Fraction



(b) Performance on class Sorting

Figure 3.8.: Combining Seeding, "Best" Hybridization, and Uniform Fitness Inheritance.

4. Guidance

Modern software systems are complex, and multiple analysis techniques are required to form an overall and detailed enough picture. Even if someone only considers a simple procedure, and he focuses only on the functional properties, it is known that the control-flow graph alone can fail to provide enough guidance towards a particular goal [FK96]. The object-oriented paradigm introduces even more subtle dependencies, since the behavior of an object might also depends on its state.

In order to provide *TestFul* with enough guidance, we use different analysis techniques, and we reward tests according to several, complementary coverage criteria. In particular, we combine a black-box criterion, presented in Section 4.1, with two white-box criteria, namely the structural coverage (Section 4.2) and the data-flow coverage (Section 4.3).

4.1. Coverage of the Behavioral Model

Methods of object oriented systems activate different behaviors depending on the internal state of the object accepting the call and on the actual values of the input parameters. Consequently, if one wants to execute a precise behavior, he must follow a precise *interaction protocol* with the system and have to both prepare the state of the object and invoke the right method with the adequate parameters. Considering all the interaction protocols for a given class, the *object behavioral model* would be obtained. This model analyzes the class from an external point of view (i.e., it is a *black-box* analysis technique).

We propose to measure the ability of tests to cover the object behavioral model, and to use this black-box information to drive evolutionary global search of *TestFul*.

The remaining part of this section first introduces ADABU, an approach to mine the object behavioral model, then presents our approach to mine and represent the behavior of classes, and finally explains how *TestFul* can leverage this information to drive the global evolutionary search.

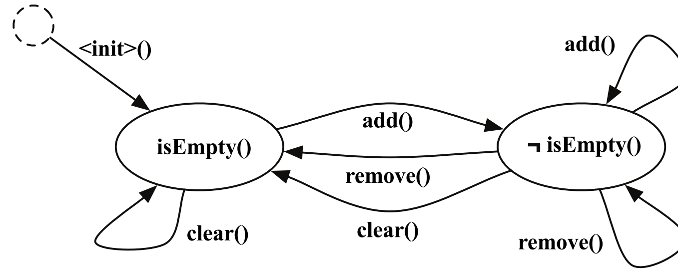


Figure 4.1.: ADABU's object behavioral model for the `Vector` class.

Mining Object Behavioral Models

Even if object behavioral models are considered important, they often are missing, incomplete, or outdated. Consequently, there are several approaches that reassemble the object behavioral model using static and/or dynamic analysis.

ADABU [DLWZ06] monitors the execution of tests on the system to mine object behaviors. Its goal is to provide the user with a model of the normal behaviors of the system. As formalism, it uses finite state machines, where nodes represent "meaningful" states of the system and edges depict transitions between states. The former are obtained by applying simple abstraction functions to observable properties, while the latter are just method invocations. For example, Figure 4.1 reports the behavioral model of the class `Vector`. It comprises two states: *empty* and *not empty*. They are directly deducted from the observer method `isEmpty`. In this case, the same two states can be obtained by applying the ADABU's abstraction functions to the observer method `size`: $size < 0$ is meaningless, $size = 0$ corresponds to *empty*, and $size > 0$ is *not empty*. Edges indicate the effects of method invocations, and pinpoint the new states that the object might assume. It is worth to notice that the model is not a detailed specification of the system, as it does not explicit, for example, neither the object returned nor the final state of the object when the *remove* operation is invoked in the *not empty* state. However, the developer can analyze it to understand the overall behavior of the class and to spot high-level errors (for example, the presence of an *add* edge headed towards the *empty* state would be a symptom of an error).

Towards a more complete Behavioral Model

We note that the model proposed in ADABU is oversimplified, and it might fail to properly render the behaviors of systems. In particular, we

highlight two main limitations: it might fail to recognize some object states, and it does not consider the parameters of methods.

Some states cannot be recognized by only considering a single property and applying a simple abstraction. For example, let us introduce class `BoundSet`, which implements a set with a limited number of elements. In order to recognize whether a `BoundSet` instance is *full*, the `size` property and the ADABU's abstraction functions are not enough; instead one have to also consider the `capacity` property and adopt a more sophisticated abstraction function that compares the two properties.

The object behavioral model used in ADABU only considers the state of the object, and neglects the used parameters. On the contrary, the behavior of object-oriented systems depends on both the internal state and the given parameters. For example, considering class `BoundSet`, method `insert` behaves differently if the provided parameter is already in the set or not.

For these reasons, we provide more advanced abstraction functions, and we also consider the actual parameters used in method invocations.

We provide the user with several *parametric* abstraction functions, which allows him to abstract both methods and states of the class, by considering the provided parameters.

Boolean accepts a *boolean* expression, and creates two states —*true* and *false*— depending on the value of the expression.

Reference accepts an expression whose result is a reference, and discriminate whether it is *null* or *non-null*.

String requires an expression with type *string*, and creates the following states: *null*, *empty*, and *not empty*.

Number accepts an expression that generate a number (natural or floating point). The user can also provide some expressions as parameters, which are used as yardstick (if none is provided, the default "0" value is used). The abstraction function returns the interval that contains the number returned by the expression. For example, the $this.size()$ property of the class `BoundSet`, when no parameters are provided, is abstracted in three states[1]: $this.size() < 0$, $this.size() = 0$, and $this.size() > 0$. Conversely, we shown that if one provides as parameter both 0 and $this.capacity()$, all the interesting states are detected: $this.size() < 0$, $this.size() = 0$,

[1]In this case, the *Number* abstraction function behaves like the ADABU's abstraction function.

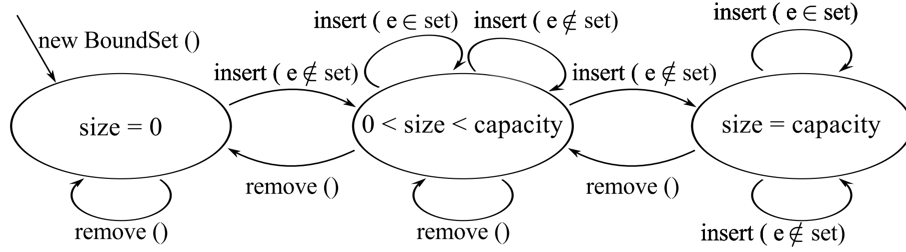


Figure 4.2.: Behavioral model of class `BoundSet`.

$0 < \mathit{this.size}() < \mathit{this.capacity}()$, $\mathit{this.size}() = \mathit{this.capacity}()$, and $\mathit{this.size}() > \mathit{this.capacity}()$.

Membership accepts an expression whose result is a reference, and, as parameters, some expressions that return a collection of elements (e.g., `Iterable`s, `Iterator`s, or *array*s). As result, indicates if the provided reference is included in some parameters, or not.

Before and after executing each operation of the test, we analyze the state of the system through observers. Abstract states —that summarize the influence of the internal state on the behavior of the system— are obtained by using parametric abstraction functions. These accept as input an expression —in which the user specifies how to inspect the system by invoking its *observers*— and provide as output the abstract state of the system. For example, if we consider the `BoundSet` class, the user can specify to abstract the expression $\mathit{this.size}()$ using the *number* abstraction function, by comparing its value against 0 and the expression $\mathit{this.capacity}()$ (i.e., they are provided as parameters). Consequently, the following abstract states are identified: $(\mathit{size} < 0)$, $(\mathit{size} = 0)$, $(0 < \mathit{size} < \mathit{capacity})$, $(\mathit{size} = \mathit{capacity})$, and $(\mathit{size} > \mathit{capacity})$. In this example, the first and the last states are meaningless, but the other three are respectively the *empty*, *non-empty*, and *full* abstract states (Figure 4.2).

Edges between nodes render the effect of a method invocation on the abstract state. For example, a `remove` from a *full* set makes the abstract state change to *non-empty*. Moreover, the behavior of methods can also depend on provided actual parameters. Our approach allows expressions to reference the method's parameters through special variables (the ith parameter is accessible through the p_i variable). For example, the method `insert` behaves differently if the parameter is a duplicate element. For this reason, the user can provide the expression $p0$ to consider the (first) argument of the method invocation, and choose the

membership abstraction function with *this* as parameter. In this way, two versions of the `insert` method are created, depending on whether the value being inserted already belongs to the set or not.

Behavioral coverage

The object behavioral model explicits the different behaviors that methods can have, according to the current state of the object and the used actual parameters. For example, method `insert` behaves differently if the `BoundSet` is *empty*, *non-empty*, or *full*, and if the element being inserted is already present in the set or not. Consequently, we reward tests according to their ability to cover the edges of the behavioral model. In fact, different edges represent a different behavior of the class. For example, a test that invokes method `add` both on the *empty* state and the *not empty* state is more worth than a test that only performs the first method call.

In order to measure the behavioral coverage, we monitor the system while each test is executed, and we infer the behavioral model that corresponds to the interactions exercised. Then, we reward each test according to the number of different behaviors it is able to exercise (i.e., the number of edges), and we calculate its **behavioral coverage** (cov_{beh}). This information directly contributes to the fitness function.

Our approach also establishes a feedback loop between the construction of the behavioral model and the test generation. Behavioral models are inferred by monitoring the execution of tests, and tests are rewarded according to the completeness of the inferred models. Hence, as soon as a test exercises a new feature, its behavioral model becomes more complete, its behavioral coverage increases, and the evolutionary engine will use that test more often as a basis for generating new tests. This leads to the creation of (i) a test able to exercise all the features of the system, and (ii) a model that completely renders them in a format that can be read by a human being. To the best of our knowledge, only Xie [Xie06] establishes a similar feedback loop between model inference and test generation. Our approach differs from Xie's one on the underlying model: he focuses on the contracts of classes [Mey92], leverages on Daikon [ECGN99] to infer them automatically, and uses a specification-based test-generation approach, instead we adopt the object behavioral model.

4.1.1. Empirical Evaluation

Chapter 2 shows how we used the information contained in the *behavioral coverage* to drive the holistic evolutionary algorithm. Our preliminary empirical evaluation (see Section 2.5) confirms that the evolutionary algorithm can leverage this information to select tests, fruitfully recombine them, and converge towards an optimal test.

We were able to leverage the abstraction function to successfully extract the behavioral model from several classes of our benchmark:

State Machine. The behavioral model of this class is simply the state machine implemented by the class. The expression `this.getState` allows one to determine the current state of an object. Then, we applied the *Number* abstraction function, with the *id* of the states as parameters, to achieve the abstract states.

Coffee Machine. We abstract the coffee machine basing on the values of methods *getTarget* and *getState*. The former is compared against two options: zero (the user has not selected any beverage) and a positive value. The state is compared with the target's value, forming the following states: {(state=0), (0<state<target), (state=target), (state>target)}.

Fraction. The behavior of a fraction object heavily depends on its internal state (i.e., the number it represents). Obviously several visible states are equivalent (e.g., $\frac{1}{1} = \frac{2}{2}$), abstractions must reassemble the behavioral model correctly. We analyze the state of the object by applying the *Numerical* abstraction function to the following observers:

floatValue: we compare the floating point value of the fraction (f) against -1, 0, 1. In this way we produce the following states:
$\{ (f < -1), (f = -1), (-1 < f < 0), (f = 0), (0 < f < 1), (f = 1), (f > 1)\}$.

denominator: the denominator of a fraction can only assume positive values or zero by construction. Our abstraction discriminates these two cases by producing two abstract states:
$\{(den = 0), (den > 0)\}$.

numerator: we compare the numerator with the value of the denominator, its opposite, and 0. Accordingly, we generate the following states:
$\{(num < -den), (num = -den), (-den < num < 0), (num = 0), (0 < num < den), (num = den), (num > den)\}$.

The overall behavioral model of this class is the combination of these states. Some combinations are infeasible (e.g., if the numerator is less

than the denominator, the corresponding float value cannot be greater than 1), while others have special meaning: for example, when the denominator is zero, the fraction either represents a *not a number* (NaN) or an infinite value, depending on whether the numerator is zero.

Fault-detection ability of the Behavioral Coverage

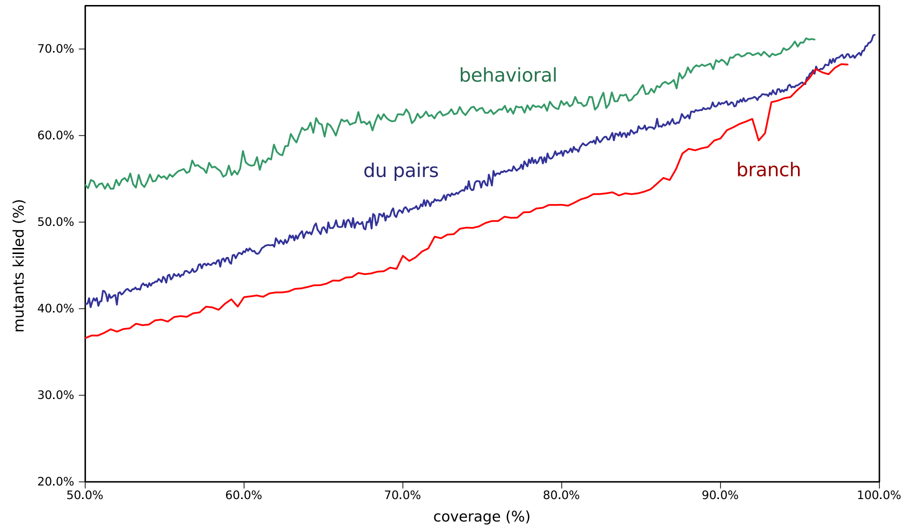
However —since the *behavioral coverage* constitutes a novel coverage criterion— we must investigate its fault-detection ability.

We select as benchmark the class `Fraction`, which is a well-designed class with a non-trivial behavior. We compare the *behavioral coverage* against two widely accepted coverage criteria: the *branch* coverage and the *all du-pairs* coverage. They reward tests according to their coverage of the control-flow and data-flow graphs, respectively.

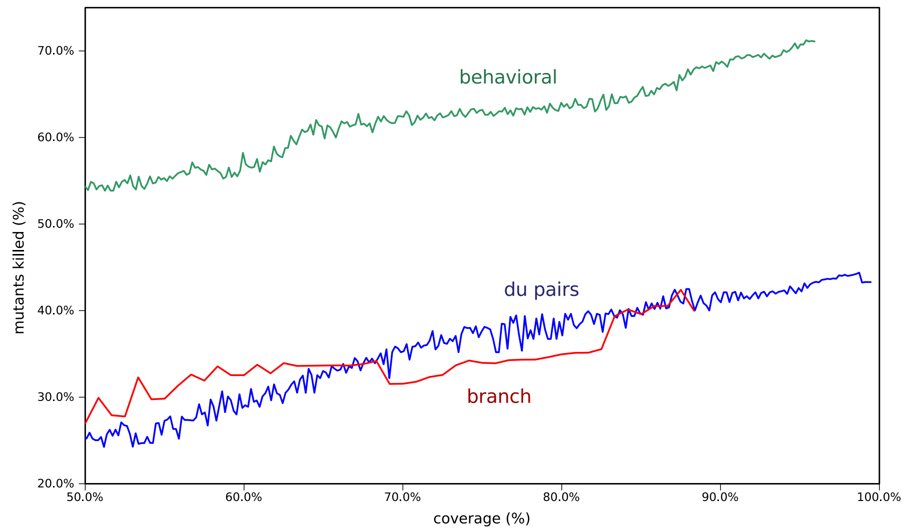
To perform a good comparison we generate an uniformly distributed set of tests for each considered criterion. We focus on the upper 50% coverage, and we generate 10 tests for each 0.1% coverage (e.g., we have ten tests between 87.6% and 87.7%). To generate tests without introducing any bias, we used three versions of *TestFul*, each guided by the considered criterion only. Even if this limited guidance might prevent *TestFul* to generate tests with the complete coverage of the system, it allows us to compare the criteria and draw important conclusions. Then, we applied the mutation analysis (briefly summarized in Appendix A) to measure the fault detection ability of each test (i.e., its *mutation score*). Andrews et al. [ABL05, ABLN06] show that this technique is able to replicate the typical mistakes that developers do, and it is adequate to compare two different coverage criteria.

Figure 4.3(a) reports the result of the experiment, and for each considered criterion compares the coverage level (reported on the x-axis) against the mutants killed (y-axis). The low mutation score (below 70%) is due to the presence of equivalent mutants. We were not able to prune them out because the tool we used applies *mutation schemata* directly on the bytecode, and it does not allow us to inspect each mutant individually. However, this is recognized to be a common problem [DLS78, BS79]. It does not undermine the comparison since it influences in the same manner all the three coverage criteria, it does not undermine the comparison. The graph shows the good fault-detection ability of tests with an high level of *behavioral coverage*, which outperforms both the *branch coverage* and the *du-pairs coverage* criteria. Consequently, it seems reasonable to judge the quality of tests according to their ability to exercise all the user-visible behaviors.

Interestingly, a previous version of the class `Fraction` was *not correct*



(a) Tests generated for the complete class.



(b) Tests generated for the incomplete class.

Figure 4.3.: Mutation Analysis on class `Fraction`.

Algorithm 2 Branch selection example.

```
1: if a > 5 then
2:     if b < 0 then
3:         do something
4:     else
5:         do something
6:     end if
7: else
8:     do something
9: end if
```

since it failed to handle fractions with a zero denominator. In particular, the code require to handle those special states was completely missing, and it was added in later releases. This allow us also to study the ability of the considered coverage criteria to overcome errors presents in the class and assign the correct quality to tests. Consequently, we perform another empirical study. For each criterion, we select tests according to their coverage on the *wrong* class, and we calculate their mutation score on the *correct* class. As initially expected, white-box criteria perform poorly. They were not able to correctly judge the tests because the code to handle the special cases was completely missing, hence tests with a high coverage have a low mutation score. Instead, the *behavioral coverage* analyzes the system using a *black-box* technique: it was not misguided by the missing code, it was able to correctly judge tests, and it selected tests with a higher mutation score, outperforming the *white-box* techniques.

4.2. Coverage of the Control-Flow Graph

The global search seeks for a test able to put objects in interesting states. In order to recognize them, *TestFul* uses as heuristic the level of structural coverage that each test achieves. The higher it is, the more the test is able to exercise the involved classes, thus the more likely it puts objects in interesting states. In particular, we adopt both *statement* and *branch* coverages, and we add cov_{stmt} and cov_{branch} components to the fitness function. Note that the former is able to detect behaviors not covered by the latter, such as handling an exception.

4.2.1. Local Search

To apply the step of local search presented in Section 3.1, we identify as targets the branches that are reachable, but not exercised yet.

A branch b is said to be reachable, but not exercised, when it belongs to a condition already evaluated and b has never been taken. For example, let us consider the program fragment of Algorithm 2 together with a simple test that exercises it with `a` and `b` set to 2 and 3, respectively. The condition at line 1 is evaluated, the *false* branch is taken, and line 8 is executed. The *true* branch (lines 2-6) is reachable, but not exercised. Conversely, since the condition at line 2 is not evaluated, its branches (lines 3 and 5) are neither reachable nor exercised.

4.2.2. Empirical Evaluation

We compared *TestFul* against some promising search-based approaches able to work on stateful systems, namely *jAutoTest* (a version of AutoTest [LCO+07], developed for Java), *randoop* [PLEB07], and *etoc* [Ton04]. *jAutoTest* mainly differs from *AutoTest* in the ability to generate a synopsis of the executed test, usable for regression testing. It monitors the random execution of the system, storing those operations that exercise uncovered statements or branches. Consequently, the synopsis achieves the same level of coverage as the whole random execution.

We chose a benchmark of 15 classes from the project listed in Section 2.4: `Array Partition`, `Binary Heap`, `Binary Search Tree`, `Coffee Vending Machine`, `Doubly Linked List`, `Disjoint Set` (both the *reference* and the *fast* versions), `Fraction` (from the *Apache Commons Math* project), `State Machine` (both the *Simple* and the *Hard* versions), `Red Black Tree`, `Sorting Stack` (both the version using an *Array* and that using a *List*), and `Vector`.

Among them, *Red Black Tree* shares the same data structure as `java.utils.TreeMap`, used in several related works as benchmark. *Hard State Machine* extends the *Simple State Machine* used in [MS07] by introducing a sink error state that hinders the generation of tests able to reach the target goal.

To judge the effectiveness of each considered approach, we set the time limit for the test generation to 5, 10, 20, 40 minutes: the more time is available, the better the resulting test should be. For each combination of classes, tools, and time limits, we performed ten runs, generating each time a test. To calculate the statement coverage and the branch coverage, we replayed each test using *cobertura* [cob], a third party code coverage reporting tool. To achieve more accurate results, the comparison was made on the average value over the ten runs. We evaluated the quality in terms of (i) statement coverage, (ii) branch coverage, and (iii) size of generated tests.

Our experiments stressed the tools heavily, requiring a total of 750

hours of CPU-time on an Intel Xeon E5530@2.40GHz with 6 gigabytes of RAM. Instead of using toy examples, we preferred to work with real classes, taken from independent parties, but this caused some problems. Because of the prototypical level of the tools, given the 600 simulation runs for each tool, *TestFul*, *jAutoTest*, *randoop*, and *etoc* successfully completed 598, 595, 235, and 583 runs respectively.

Randoop was not able to handle two classes of our benchmark, namely *Stack Array* and *Stack List*, and terminated its execution throwing an exception. Since these problems are mainly related to errors in the implementation in the tool, we did not penalize the approach by excluding these two classes while we calculate its average performance. Moreover, the memory required by *randoop* increased with the duration of the runs, and we were not able to successfully generate and replay a test with runs longer than ten minutes. With longer time limits, either the tool crashed running out of memory, or the replay failed due to the impossibility of compiling the generated test (the compilation ran out of memory, requiring more than 5 gigabytes).

Similarly, we were not able to replay directly tests created by *etoc* for the *Stack Array* class. After investigating the problem, we discovered that some of the methods of that class declare to throw an exception, but that was ignored by the generated jUnit tests. In this case, we solved this naive error by manually modifying the generated tests.

In order to evaluate the approaches more than the tools, we discarded the results when we got errors (e.g., out of memory[2] or non-termination[3]) either in the test generation or in its replay. Thus, we calculated the average performance for each tool on each class within the time limit by only considering successful runs. Finally, for each approach and for each time limit, we calculated an abridged version of the results by calculating the average performance over all considered classes.

By analyzing the structural coverage achieved on each class, we noticed that often a complete coverage was not achieved by any tool. We investigated this phenomenon, and we found that it appears only on classes taken from the independent repository. This can be explained with two reasons.

- The classes contain some unreachable code, such as some private utility methods never used in the class.

[2]We used a heap size of 1,800Mb and 5,000Mb respectively for test generation and test replay (including its compilation).

[3]If an approach required 30 minutes more than the time limit to generate a test, we terminated the run marking it as *non-terminated*. Similarly, if a test replay required more than 45 minutes, we marked the run as *non-terminated*.

- The classes declare protected or friendly methods never invoked explicitly. Note that all approaches behave as external users of the class, and thus they only considered public methods.

Since these two facts impacted on all the tools the same way, we decided not to modify our benchmark.

Experimental Results

This section provides a summary of the experiments performed[4]. For each time limit (i.e., 5, 10, 20, and 40 minutes) we run the simulations using all the four tools. Table 4.1 reports the average performance (μ) and the related standard error (s) of the size of generated tests (Lines Of Code), statement coverage, and branch coverage. The last two values are also plotted in Figure 4.4.

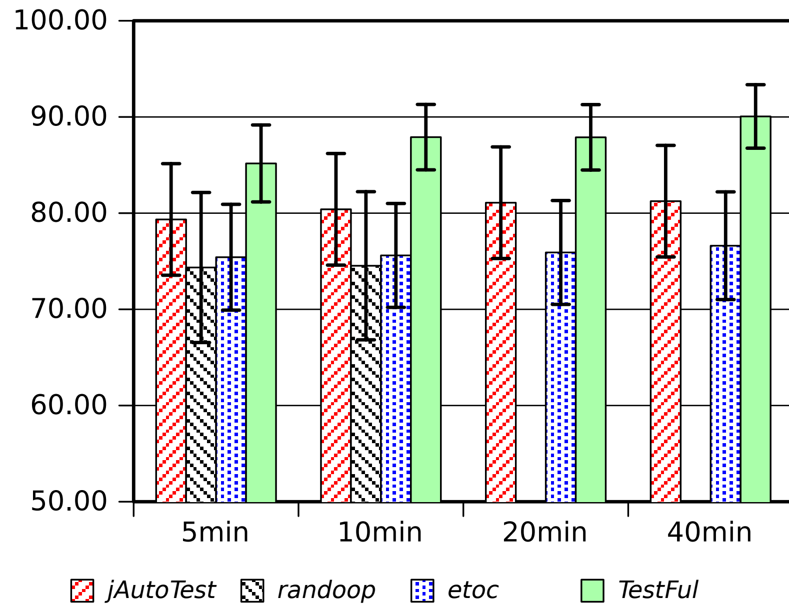
If we consider the mean of structural coverages (both statement and branch coverage), *TestFul* outperforms the other approaches for all time limits and confirms its ability to generate good tests by working both at class and method level. Moreover, the more time is available, the bigger the gap between *TestFul* and the other tools is.

If we consider the standard error of the mean, we can recognize two groups: one formed by *jAutoTest* and *randoop*, and the other by *etoc* and *TestFul*. The first group has a higher standard error, which remains constant even with long runs. The second group has a minor standard error. It is also important to note that this decreases with long runs when *TestFul* is employed. This can be explained by considering the way the search space is explored: the first group uses a blind search, while the second uses some kind of guidance. The presence of guidance ensures more repeatable results, thus lowering the standard error. Moreover, *TestFul*'s error decreases as runs became longer, which suggests that it is converging towards a (sub-)optimal solution.

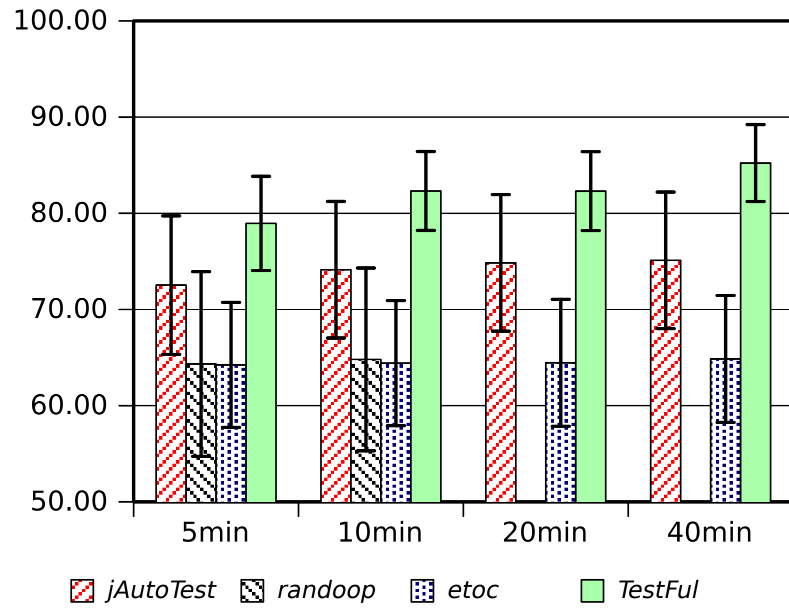
As far as the size of generated tests is concerned, *TestFul* creates a test suite smaller than *randoop* and *jAutoTest*, but bigger than *etoc*. On top of that, the tests generated by *TestFul* are more suitable for regression testing, since the replay time is comparable with the time required to replay *etoc*'s test (both require a few seconds). In contrast, *jAutoTest* and *randoop* generated huge tests, difficult to use in the form they are for regression testing (they would require several minutes to run).

It is also interesting to analyze the possible relationship between the size of the tests and the length of the runs for the four tools. *Randoop*

[4]The complete set of results is available at
`http://home.dei.polimi.it/miraz/testful/icst10`.



(a) Average Statement Coverage.



(b) Average Branch Coverage.

Figure 4.4.: Average structural coverage.

Table 4.1.: Performance.

(a) 5-minute runs.

Tool	**Size of Tests** $\mu(s)$ LOC	**Statement Coverage** $\mu(s)$ %	**Branch Coverage** $\mu(s)$ %
jAutoTest	315,448 *(133,760)*	79.3 *(5.8)*	72.5 *(7.2)*
randoop	3,835,150 *(509,377)*	74.3 *(7.8)*	64.3 *(9.6)*
etoc	83 *(17)*	76.5 *(5.5)*	65.8 *(6.5)*
TestFul	23,634 *(12,196)*	85.2 *(4.0)*	79.0 *(4.9)*

(b) 10-minute runs.

Tool	**Size of Tests** $\mu(s)$ LOC	**Statement Coverage** $\mu(s)$ %	**Branch Coverage** $\mu(s)$ %
jAutoTest	357,130 *(153,461)*	80.4 *(5.8)*	74.1 *(7.1)*
randoop	7,823,575 *(984,493)*	74.5 *(7.7)*	64.8 *(9.5)*
etoc	85 *(17)*	76.9 *(5.4)*	66.3 *(6.5)*
TestFul	16,694 *(9,160)*	87.9 *(3.4)*	82.3 *(4.1)*

(c) 20-minute runs.

Tool	**Size of Tests** $\mu(s)$ LOC	**Statement Coverage** $\mu(s)$ %	**Branch Coverage** $\mu(s)$ %
jAutoTest	367,860 *(164,300)*	81.1 *(5.8)*	74.8 *(7.1)*
etoc	86 *(18)*	77.0 *(5.4)*	66.3 *(6.6)*
TestFul	9,600 *(6,208)*	87.9 *(3.4)*	82.3 *(4.1)*

(d) 40-minute runs.

Tool	**Size of Tests** $\mu(s)$ LOC	**Statement Coverage** $\mu(s)$ %	**Branch Coverage** $\mu(s)$ %
jAutoTest	368,662 *(169,718)*	81.2 *(5.8)*	75.1 *(7.1)*
etoc	88 *(20)*	77.3 *(5.6)*	66.1 *(6.6)*
TestFul	8,054 *(5,845)*	90.1 *(3.3)*	85.2 *(4.0)*

shows a direct relationship: the size of 10-minute runs are slightly more than twice the size of 5-minute runs. This phenomenon prevents us to have runs longer than 10 minutes. The size of the tests generated by *etoc* and *jAutoTest* are almost stable. For *jAutoTest*, this can be explained by considering that it only emits the synopsis of the whole execution. *TestFul* instead generates shorter tests with longer runs. To understand

this, we should consider the problem of *bloat* [LP97]. Evolutionary algorithms using variable-length elements tend to introduce sequences of useless genetic material, called *introns*. The presence of introns eases the recombination and the mutation of elements, since the probability to split or alter useful elements decreases. Our fitness function contains a component aimed to minimize the length of individuals, but its overall contribution is limited. We might increase its weight, but acting in this way we would decrease the exploration of the solution space, pushing the search towards shorter tests, and decreasing the capability of *TestFul* to generate good tests. Since the size of the result was acceptable, we decided to leave the fitness function unchanged; the user may always adopt some analysis techniques (e.g., def-use analysis, slicing, or delta debugging) to produce a smaller tests, as shown in [LOZ+07].

Figure 4.5 compares branch coverage against time limits to understand whether the approaches achieve better results with longer runs (statement coverage shows a similar trend). The more the available time increases, the more thoroughly *TestFul* exercises the class under test. *jAutoTest* shows a similar behavior, but with a much less gain. Conversely, the performances of the other tools remain almost stable. This phenomenon can be explained by considering that *jAutoTest* and *TestFul*

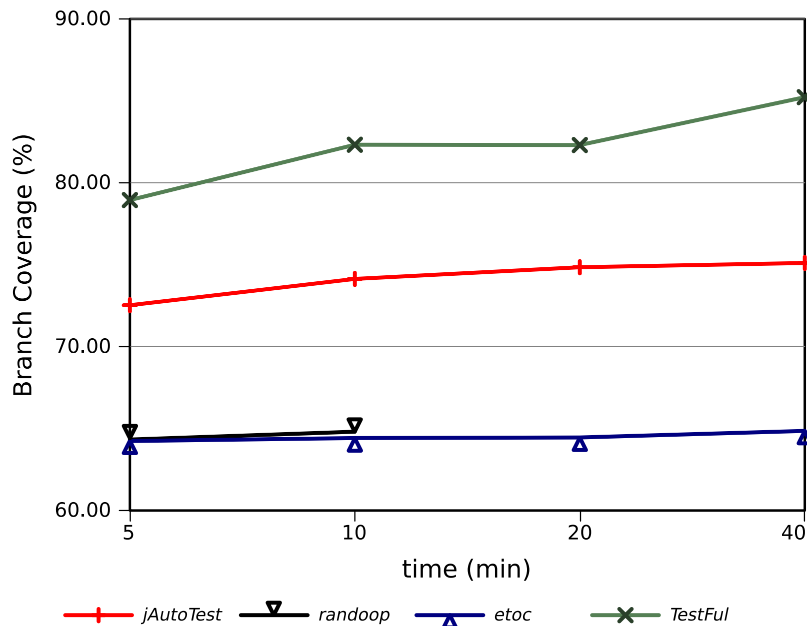


Figure 4.5.: Branch Coverage vs. time limit.

have a similar internal test representation. This representation fosters the evolution of objects' states, thus longer runs reach more complex configurations and achieve higher structural coverages.

4.3. Coverage of the Data-Flow Graph

Many objects have a hidden internal state, which heavily influences the behavior of methods. For this reason, tests for object-oriented systems are often rewarded by considering the coverage of the data-flow graph, which correctly relates methods that cooperate by exchanging data (e.g., through objects' fields).

When a statement assigns a value to a variable v, that statement is said to **define** the value of v. Similarly, a **use** is a statement that uses the value of a variable. If there are two statements s_d and s_u in which the first one defines the value of a variable v ($def_v = \{s_d\}$), and the second one uses the value of v ($use_v = \{s_u\}$), and if a path exists from s_d to s_u in which v is not re-defined (*killed*), then there is an association between the definition and the use, and they form a *du-pair* (definition-use pair).

Several coverage criteria use this concept [RW85]. In particular, *TestFul* adopts the *all du-pairs* coverage criterion, which requires that the tests exercise all feasible du-pairs. In this way, if an erroneous value is produced and assigned to a variable, it should also affect a statement that uses that value. In *TestFul* this corresponds to extending the fitness function with $cov_{du}(t)$, and to reward tests according to the number of exercised du-pairs. The higher the value is, the more the test exercises the class, and the more likely it will be selected by the *TestFul*'s evolutionary algorithm.

Note that our work does not need to calculate the whole set of possible du-pairs, which is an extremely expensive operation especially when performed inter-procedurally. In fact, *TestFul* only requires the number of du-pairs that have been exercised to determine whether a test exercises the class more than another one. This is done by instrumenting the class under test (a) to track the definitions of each variable and (b) to form a du-pair when the variables are used.

Sometimes the uses in "normal" computational statements (*c-use*) and those in predicates (*p-use*) are considered to be different; in this case, one should count a p-use for each branch that starts from the predicate [RW85]. For example, consider the simple example reported in Algorithm 3. Variable v has a single definition (line 1) and a single use (line 2), hence there is a single def-use pair. However, the use happens in a conditional statement with two outgoing branches, and there are two

Algorithm 3 P-Use example.

1: $v \leftarrow read()$
2: **if** $v > 0$ **then**
3: $do_something$
4: **else**
5: $do_something_else$
6: **end if**

p-uses: one for the "true" branch and one for the "false" branch.

Using these definitions, *all c-use* and *all p-use* adequacy and coverage criteria have been defined: they require to exercise all the definitions that reach all the computational and predicate uses, respectively. It is worth to note that the *all du-pairs* adequacy criterion subsumes the *all c-use* one, while it does not subsume the *all p-use* criterion. The *all p-use* criterion relates uses with the branch executes, hence "false" and "true" uses of a conditional statement are treated as different uses. For this reason, *TestFul* allows the user to judge tests also according to the *all p-use* criterion, and extend the fitness function with the cov_{pu} component.

4.3.1. Local Search

Data-flow information can be used in two different ways to perform the step of local search we presented in Section 3.1. On one side, data-flow information can be used to identify as targets those def-use pairs that are feasible on the system, but that are not executed by any test. On the other side, we can leverage data-flow information to refine the identified targets by removing those that are infeasible.

Improving the data-flow coverage

In order to improve the def-use coverage, one could focus on reaching a new def-use pair and improve the *all du-pairs* coverage. To solve this problem, one should enforce a complex constraint: the targeted definition and the targeted use shall be executed, and the definition shall not be killed before reaching the use. However, the local search cannot be asked to solve such complex constraints, as it would probably fail in addressing the three parts of the constraint.

Instead, we use the data-flow information to identify as targets uncovered *p-use*s. Since chasing new definition-use pairs is too complex for the local search, we want to make the existing pairs reach all the possible branches, so to augment the *p-use* coverage criterion (Obviously, for each

definition-use pair we only consider the branches of the condition that contains the use). To this end, we collect and analyze the *p-use* coverage, composed by triples in the form $(def, use, branch)$. Obviously, there is a relationship between uses and branches, as they refer to the same condition. We analyze the def-use pairs and the branch separately, so to spot whether definitions that reach a p-use fail to enter all the branches of the related condition. Finally, we identify these missing $(def, use, branch)$ as *p-uses* targets.

For example, consider the condition at line 6 in Algorithm 4: it contains a predicate-use of the variable v. Analyzing the code, it is clear that definitions at lines 2 and 4 reach that statement, hence the to fulfill the *p-use* adequacy criterion one should cover the following p-uses:

$$\{(2, 6, true), (2, 6, false), (4, 6, true), (4, 6, false)\}$$

Let us suppose that a test only covers the first one. *TestFul* analyzes the *p-use* coverage and identifies that the test is able to pair definition 2 with p-use 6, but this pair fails to execute the false branch. Accordingly, *TestFul* prompts as target the p-use triple: $(2, 6, false)$. This target is added to those identified by the control-flow analysis. In the considered example, the *false* branch of condition at line 1 is never executed, and it is prompted as a *branch* target. Suppose that this target is selected and successfully reached by the local search (indeed, *branch* targets are easier and hence more likely to be selected than *data-flow* targets). Suppose also that the resulting test have an improved p-use coverage, composed by the triples:

$$\{(2, 6, true), (4, 6, false)\}$$

TestFul, analyzing the coverage, reckons that the test pairs definitions 2 and 4 with p-use 6. Analyzing the branch covered by each pair, *TestFul* understands that there are missing p-use entries. *TestFul* identifies that definitions 2 and 4 reach the condition, but they fail to reach all the

Algorithm 4 P-Use selection example.

1: **if** *condition* **then**
2: $\quad v \leftarrow v1$
3: **else**
4: $\quad v \leftarrow v2$
5: **end if**
6: **if** v **then**
7: $\quad$ *do something*
8: **end if**

branches of that condition. In fact, definition 2 enters the *true* branch, but misses the *false* one. Similarly, definition 4 executes the *false* branch, but leaves out the *true* one. Accordingly, *TestFul* identifies as *p-use* targets the missing triples: $(2, 6, false)$ and $(4, 6, true)$.

Identification of unfeasible targets

The list of targets might contain two types of targets: some aiming to improve the *branch* coverage, other to higher the *p-use* coverage. The local search then selects one of these targets, and searches for a test able to solve it (i.e., either reach the selected branch or exercise the given p-use). If a solution is found, it is merged into a test, so that the evolutionary algorithm of *TestFul* can reuse it.

To guarantee that a high efficiency of the approach, it is mandatory to avoid spending computational effort on targets not easily reachable from the current test. For this reason, we leverage control-flow and data-flow information to detect whether a target is not feasible or if a target depends on other ones. During the instrumentation of the classes, *TestFul* saves the variables, the constants, and the comparison that compose each conditional statement. Since the instrumentation is made on the Java's bytecode directly, only "simple" conditions are found, composed of two values (variables or constants) and a comparison operator. This information is used a run-time to understand which values can reach each conditional statements, and to determine whether one of its branches can be reached or not. Given a condition in the form $a \oplus b$, there can be three different cases:

- Both a and b are constants.
- a is a variable and b is a constant (or vice versa).
- Both a and b are variables.

The first case, notwithstanding it can be automatically resolved by an optimizing compiler, makes some target unreachable at all. In fact, the outcome of the condition is fixed, as there is no way to change the values and the comparison operator. Even if *TestFul* does not prune the unreachable code during the instrumentation, it is able to detect if such unfeasible branch is targeted. Since *TestFul* makes available at run-time the information regarding the condition, it have the values being used in the condition (in this case, they are constant), hence it can pre-calculate the outcome and detect unfeasible branches.

To handle the second case, *TestFul* verifies if the variable can only assume a limited set of values, and if it is possible to pre-determine

the outcome of the condition. For this reason, it analyzes the data-flow information to collect the definitions of the variable (a) that reach the conditional statement. If all these statements define a with a constant value, it is possible to determine the values the variable can assume, and verify if the target can be reached or not. In the latter case, the target is marked as unreachable (starting from the current test). However, a can be defined without using any constant value (e.g., by invoking a function). In this case, we cannot limit the values the variable can assume (we say that the variable is "*free*" to assume any value).

The third case is an extension of the second case. In order to predetermine the outcome of the condition, both variables must be initialized with constant values.

For example, we can consider the classical use of a flag variable reported in Algorithm 5.

Algorithm 5 Usage of a Flag-variable.

1: $flag \leftarrow false$ ▷ Definition 1
2: **for all** $e \in elems$ **do**
3:     **if** $condition(e)$ **then**
4:         $flag \leftarrow true$ ▷ Definition 2
5:     **end if**
6: **end for**
7: **if** $flag$ **then** ▷ Use 1
8:     $target$
9: **end if**

Let us suppose that line 8 is not executed. In this case, *TestFul* selects both lines 4 and 8 as targets, with the corresponding conditions (lines 3 and 7, respectively). The first condition (Line 3) uses a free variable, hence *TestFul* correctly does not mark it as infeasible. The second condition (Line 7), instead, uses the value contained in variable `flag` (Use 1). However, the du-pairs coverage reports that *Use 1* is only reached by *Definition 1* (line 1), which sets the value of the flag with a constant value ("*false*"). This value does not satisfy the condition being addressed (line 3), which requires the flag to be "*true*". In this way, *TestFul* detects that the condition cannot be targeted directly, and it discards the corresponding target. Consequently, *TestFul* directly targets the right condition (i.e., the one at line 3), and it does not wast computational effort by applying the traditional trial-and-error process used by other evolutionary approaches [FK96].

We can use the *def-use pairs coverage* to ensure that the local search does not takes blind alleys. In some situations, a change of the test

Algorithm 6 Simple test for the class `Water`.

```
1: Double d = 212.0;
2: Water w = new Water();
3: w.setFarenheitTemperature(d);
4: w.checkStatus();
```

can make the values of the variable get closer to the target, but turning out the target itself infeasible. For example, consider a the class `Water` which allows the user to set a Fahrenheit temperature and check its status (*solid*, *liquid*, or *gas*). Suppose that the melting point is never tested, and a local search targets it by starting from the test reported in Fig. 6. The distance between the initial test and the target is $|212 - 32| = 180$. A simple improvement would be to remove operation `setFarenheitTemperature` (line 3): the constructor sets the temperature to zero degrees and the distance would become $|0 - 32| = 32$. The hill climbing algorithm would prefer the new version of the test, since it seems closer to the target. Instead, if that change is accepted, the local search loses its ability to control the temperature of the water, and the target can not be reached.

To avoid entering these blind alleys, we analyze the *du-pairs* coverage, focusing on the variables used the targeted condition. If a change makes these variables to be only defined with incompatible constant values, we discard the change since it makes the local search loose its ability to control the values of the variables.

4.3.2. Empirical Evaluation

As aforementioned, we want to: (a) investigate whether the information gathered from the data-flow graph helps *TestFul* generate better tests and (b) compare the quality of the tests produced by *TestFul* against the quality of the tests produced by means of other approaches.

To fulfill the fist goal, we selected three configurations of *TestFul* to progressively refine the fitness function used, and better understand the impact of data-flow information:

- Configuration **basic** is only driven by the statement and branch coverage criteria. This configuration does not exploit the information gathered from the data-flow graph and it is used as touchstone. The fitness function is:

$$f(t) = \langle t.length, cov_{stmt}(t), cov_{br}(t) \rangle$$

- Configuration **+du** extends the basic one with the all du-pairs coverage criterion. It also uses the data-flow graph to improve the local search and reach targets more efficiently. The fitness function becomes:

$$f(t) = \langle t.length, cov_{stmt}(t), cov_{br}(t), cov_{du}(t) \rangle$$

- Configuration **+puse** also adds the p-use coverage criterion. The fitness function becomes:

$$f(t) = \langle t.length, cov_{stmt}(t), cov_{br}(t), cov_{du}(t), cov_{puse}(t) \rangle$$

Note that *TestFul*-**basic** provides significant improvements with respect to the version presented in [MBL10]. The main differences are the integration of efficiency enhancement techniques [MLB10], a better implementation of the hill climbing algorithm, the removal of user-provided information to drive the evolutionary process, and some further minor improvements that allow *TestFul* to better deal with complex software systems (e.g., support for arrays).

To assess the quality of defined tests, we exploited the symbolic execution engine of *Java Path Finder* [PMB+08], which is a widely used tool in the testing community, to generate tests in a different way. Instead of supplying concrete inputs to the program (e.g., numbers), it provides symbolic values and uses them to execute the program (i.e., it performs a *symbolic execution*). When conditional statements are reached, *Java Path Finder* uses the symbolic values being compared to create a set of formulae —one for each branch— that contains the constraints to execute them. Traditional constraint solvers are used to obtain the actual values for the parameters. The plan was to compare these tests, and those generated by the three versions of *TestFul*, both in terms of levels of coverage and through mutation analysis. However, it is known [UOH93] that mutation analysis is an expensive techniques, as it requires to run each generated test on each mutated version of the class. In our case, the benchmark comprises more than 200 classes, each of them has more than 2'500 mutants; we have run 10 times the three versions of *TestFul*, and each of these 30 generated tests requires in average 2,5 second to run[5]. With these settings, an exhaustive mutation analysis would require $200 \cdot 2500 \cdot 10 \cdot 3 \cdot 2,5 \simeq 434$ *days*. Moreover after some first experiments,

[5]We use time-based threshold to detect mutants that make the program not terminate. It is set as five times the original execution time, as it seems a good compromise between efficiency and accuracy. However, this threshold higher the average test execution time as we experience several of non-terminating mutants.

it seemed that the tests we generated with *Java Path Finder* were only able to kill few mutants. This is why we decided to save on resources and use mutation analysis only to compare the tests generated by *TestFul* with the results published by Mouchawrab et al. [MBLD10] about the quality of tests generated by human beings for a class `OrdSet`.

Since the classes we considered had no contracts we could exploit to detect failures, we used a widely accepted heuristic [PLEB07]: there is a failure in a method invocation when it throws an unexpected exception (in case of `NullPointerException` we discarded the failure if we gave the method a `null` parameter). However, unchecked exceptions are widely used to warn the user in case of violations of method's preconditions: they are reported in the documentation (e.g., in the *javadoc*), but they are often not part of the interface (Java does not oblige one to declare when unchecked exception can be thrown). To make the heuristic work, we enriched method interfaces by adding the unchecked exceptions listed in the documentation.

To define the baseline, we ran *TestFul*-**basic**. Besides providing interesting data about the coverage achieved (Table 2), this first run also revealed several failures. Some of them are due to exceptions used to signal an invalid usage of the class not reported in the documentation. For example, if one iterates over a `Collection` while it is being modified, a `ConcurrentModificationException` is thrown. More interestingly, we were also able to find a real error in `java.util`. It is able to handle self-referring collections:

```
Collection<Object> c = new ArrayList<Object>();
c.add(c);
System.out.println(c.toString());
// prints "[(this Collection)]"
```

However, it is not able to deal with two collections that refer each other reciprocally:

```
Collection<Object> a = new ArrayList<Object>();
Collection<Object> b = new ArrayList<Object>();
a.add(b);
b.add(a);
System.out.println(a.toString());
```

In this case, method `toString` enters an infinite recursive loop terminated by the virtual machine by throwing a `StackOverflowError`. *TestFul* reported such an error automatically, while several works on the automatic generation of tests [Ton04, XMSN05, PLEB07] targeted this

package, but they did not discover it. The official Java bug database reports the error as open issue (entry 4275605).

Given the amount of unreported unchecked exceptions and failures, after running *TestFul*-**basic** we decided to run the comparison with *Java Path Finder* by disabling the fault-detection ability of the tools to speed-up the assessment. Note that the goal was to compare the test generation capabilities, and thus disabling this feature had no impact on the results.

To generate tests, we used three virtual machines with a single core of an Intel Xeon E5530@2.40GHz CPU and 1.5 gigabyte of RAM (the heap size of the java virtual machine was limited to 1 gigabyte). We generated tests for each class using both *Java Path Finder* and the three configurations of *TestFul*. We gave each tool 30 minutes of CPU-time. Since *TestFul*'s performance depends on a random sequence of numbers, we repeated its experiments ten times to achieve more accurate results, and the comparison we made is on the mean values. In contrast, *Java Path Finder* is deterministic.

Experimental Results

The results obtained by carrying out the experiments described in the previous section can be organized in two main groups:

- the first group containing the *coverage level* cover both the impact of data-flow information on the tests generated through *TestFul* and the comparison with *Java Path Finder*;
- the second group about *mutation analysis* provide interesting results about the quality of the tests generated with *TestFul* with respect to those obtained through other approaches.

Structural and Data-Flow Coverage The inclusion of data-flow information can either improve or worsen the performance of *TestFul*.

On one side, we have to consider that tracking data-flow coverage requires that the tool to collect more data from each test execution. Tracking du-pairs and p-uses is far more complex (i.e., it requires more time) than tracking the structural coverage, hence there is a chance to lower the overall performance. To quantify the slow-down imposed by the deeper instrumentation, we considered the number of method invocations performed by the three configurations of *TestFul* in their 30-minute runs. On average, the **basic** configuration of *TestFul* performs 5,000,000 method invocations, while the +**du** and +**puse** ones only 3,850,000 (23% less) and 3,750,000 (25% less) invocations, respectively. This confirms

our hypothesis that the instrumentation required to track the data-flow coverage can limit the performance of *TestFul*.

On the other side, *TestFul* capitalizes the data-flow information to improve the local search, and this should improve its performance.

For these reasons, we analyzed the average structural and data-flow coverage of the tests generated by each configuration of *TestFul* and by the symbolic execution engine of *Java Path Finder*.

Tables 4.2 and 4.3 report the average structural coverage, and show the statement and branch coverages respectively. They also show the sample standard deviation (std). All the configurations of *TestFul* outperform the symbolic execution engine of *Java Path Finder* on all the projects except the *State Machine*. Note that we picked this project from [MS07], and we expected symbolic execution approaches to work well on these classes. As for the three configurations of *TestFul*, we can note that the average structural coverage does not decreases when the data-flow coverage is monitored, albeit it allows *TestFul* to perform less method invocations. Conversely, the data-flow information allows *TestFul* to increase its structural coverage on complex classes, and in particular it fills the gap with the symbolic execution engines on *State Machine*.

Tables 4.4 and 4.5 consider the coverage on the data-flow graph, and report the average (and the sample standard deviation) number of covered def-use pairs and predicate-uses, respectively. Once again, the **basic** configuration of *TestFul* outperforms the symbolic execution engine of *Java Path Finder* on all the projects, but *State Machine* and *Stack*. The reasons for the *State Machine* have been motivated above; those for the *Stack* can be explained by considering that symbolic execution engines target the all-paths coverage, hence they are able to exercise more the data-flow graph than the **basic** configuration of *TestFul*. Instead, it seems that the ability of *TestFul* to reach a high level of structural coverage allows it to have a good coverage also on the data-flow graph (albeit with a huge standard deviation). Moreover, if we make *TestFul* consider the data-flow coverage, the number of covered def-use pairs and p-uses increases (and the previous problems are solved). Both the **+du** and **+puse** configurations increase the average *du-pairs* covered and lower its standard deviation. The actual improvement varies and depends on the projects: some almost double their average data-flow coverage (e.g., *Binary Search Tree*), while others keep the same average value, but reduce the standard deviation (e.g., *Commons Math*). Finally, note that the **+du** configuration exercises several *predicate-uses*, confirming its ability to cover the data-flow graph. However, if one explicitly targets them and uses the **+puse** configuration of *TestFul*, the number of covered predicate-uses further increases.

Table 4.2.: Average Statement Coverage.

class	Statement coverage (%)						
	JPF	**basic**	**(std)**	**+du**	**(std)**	**+puse**	**(std)**
Array Partition	93	100	(0)	100	(0)	98	(6)
Binary Heap	6	98	(5)	100	(0)	100	(0)
Binary Search Tree	95	100	(0)	100	(0)	100	(0)
Commons Math	19	88	(1)	86	(0)	87	(1)
Disjoint Set	17	100	(0)	100	(0)	100	(0)
java.util	22	86	(2)	86	(1)	86	(1)
JGraphT	22	77	(3)	72	(1)	67	(2)
NanoXML Lite	16	65	(8)	57	(9)	61	(7)
OrdSet	7	95	(0)	95	(0)	95	(0)
Red-Black Tree	35	100	(0)	100	(0)	100	(0)
Roops	6	94	(2)	93	(0)	94	(1)
Siena	45	96	(1)	95	(2)	95	(1)
Stack	55	97	(0)	97	(0)	97	(0)
StateMachine	64	46	(3)	68	(16)	57	(7)
Average	35	89		89		88	

Table 4.3.: Average Branch Coverage.

class	Branch coverage (%)						
	JPF	**basic**	**(std)**	**+du**	**(std)**	**+puse**	**(std)**
Array Partition	90	100	(0)	100	(0)	97	(5)
Binary Heap	0	97	(10)	100	(0)	100	(0)
Binary Search Tree	95	100	(0)	100	(0)	100	(0)
Commons Math	9	82	(2)	81	(0)	81	(1)
Disjoint Set	0	100	(0)	100	(0)	100	(0)
java.util	11	80	(2)	80	(1)	79	(1)
JGraphT	12	66	(3)	61	(1)	56	(1)
NanoXML Lite	5	51	(10)	40	(12)	44	(9)
OrdSet	4	90	(0)	90	(0)	90	(0)
Red-Black Tree	27	100	(0)	100	(0)	100	(0)
Roops	1	86	(2)	84	(0)	85	(2)
Siena	21	86	(2)	84	(3)	84	(1)
Stack	46	93	(0)	93	(0)	93	(0)
StateMachine	55	30	(3)	61	(22)	42	(7)
Average	27	83		84		82	

Table 4.4.: Average number of def-use pairs covered.

class	Number of def-use pairs covered						
	JPF	**basic**	**(std)**	**+du**	**(std)**	**+puse**	**(std)**
Array Partition	18	23	(1)	30	(0)	29	(4)
Binary Heap	0	33	(6)	42	(0)	42	(0)
BinarySearch Tree	70	74	(11)	116	(0)	116	(0)
Commons Math	882	30857	(2592)	31223	(276)	31161	
Disjoint Set	2	12	(0)	12	(0)	12	(0)
java.util	922	7215	(407)	8883	(118)	8712	(208)
JGraphT	311	11002	(683)	10411	(124)	10064	(290)
NanoXML Lite	38	179	(10)	179	(11)	182	(12)
OrdSet	1	221	(15)	285	(2)	285	(2)
Red-Black Tree	72	528	(27)	741	(40)	757	(51)
Roops	21	394	(19)	405	(11)	401	(20)
Siena	481	1479	(18)	1572	(32)	1588	(4)
Stack	59	41	(3)	62	(0)	62	(0)
StateMachine	40	21	(2)	61	(23)	34	(6)

Table 4.5.: Average number of predicate-use covered.

class	Number of predicate-use covered						
	JPF	**basic**	**(std)**	**+du**	**(std)**	**+puse**	**(std)**
Array Partition	14	15	(1)	20	(0)	20	(3)
Binary Heap	0	14	(2)	17	(1)	20	(0)
Binary Search Tree	29	29	(4)	52	(1)	53	(0)
Commons Math	122	6,732	(693)	6,363	(105)	6,517	
Disjoint Set	2	6	(0)	6	(0)	8	(0)
java.util	234	2,518	(136)	3,207	(50)	3,243	(77)
JGraphT	65	2,035	(113)	1,888	(27)	1,822	(68)
NanoXML Lite	6	46	(8)	39	(9)	42	(9)
OrdSet	0	83	(7)	109	(1)	112	(1)
Red-Black Tree	20	210	(16)	314	(13)	327	(23)
Roops	6	67	(6)	67	(2)	66	(7)
Siena	142	463	(19)	508	(27)	556	(6)
Stack	27	17	(2)	27	(1)	31	(0)
StateMachine	25	10	(1)	42	(20)	19	(3)

Mutation Analysis To investigate the effectiveness in detecting faults, that is the quality, of generated tests, we employed mutation analysis [DLS78, ABLN06]. Mutant operators generate multiple mutated version of the code of interest, each one with a single fault seeded. Tests are judged according to the number of mutated versions they are able to detect (*kill*). However, the application of mutant operators might create *equivalent mutants*, which are semantically identical to the original program, hence tests cannot kill them. For example, $a > b$ and $a \geq b$ are equivalent if a cannot be equal to b. To correctly judge the fault-detection ability of tests, one should (manually) identify and prune equivalent mutants.

Although mutation analysis requires huge computational effort, it provides a good estimation of the fault-detection abilities of tests [ABLN06], allowing one to compare different ways to generate them.

In particular, Mouchawrab et al. [MBLD10] compare the fault detection effectiveness of state testing against structural testing on classes with state-driven behavior. They performed a series of controlled experiments involving students from two universities: Carleton university (Canada) and Università del Sannio (Italy). One of the three projects[6] they use as benchmark is `OrdSet`, which manages a bounded and ordered set of integers, and provides operations to add and remove an element, and merge two sets. Even if this project comprises a single class with a limited number of lines of code, its complexity is comparable to the other two projects of their study.

Since the source code of class `OrdSet` and the mutants they used are publicly available —through the Software Infrastructure Repository [DER05]— and [MBLD10] provides sufficient data, we replicated the statistical study. However the authors did not provide the tests they used in their experiments (because of copyright issues) and thus we could only perform a comparison on the overall mutation score, while we could not compare the *non-equivalent* mutation score or verify whether the student-written tests and those generated by *TestFul* are complementary.

Mouchawrab et al. used `MuJava` to automatically generate a large set of mutants (more than 800), but they did not remove or mark the equivalent ones, hence we performed this task manually. Table 4.6 reports the average mutation score and the standard deviation of the tests reported in [MBLD10][7] and those generated by the three configurations of *TestFul*.

[6] The other two projects used in [MBLD10] manage multi-threaded applications, which are currently not supported by *TestFul*.

[7] For the sake of simplicity, we use the same notation used in [MBLD10], and we label with “code” (C) the *structural tests*, and with “state” (S) the *state tests*.

Table 4.6.: Mutation score.

Provenience	***all mutants***		***not equivalent***	
	mean	**(std)**	**mean**	**(std)**
Carleton Code	56.15%	(19.99%)		
Carleton State	50.27%	(17.20%)		
Sannio Code	70.31%	(12.69%)		
Sannio State	71.96%	(12.41%)		
TestFul basic	85.36%	(1.78%)	93.15%	(1.95%)
TestFul +du	89.32%	(0.18%)	97.50%	(0.20%)
TestFul +puse	89.28%	(0.29%)	97.46%	(0.32%)

Table 4.7.: Mutation score comparison: p-value of the statistical hypothesis testing $H_0 : mean(A) > mean(B_i)$ vs. $H_1 : mean(A) \leq mean(B_i)$ with $B :=$ { "Carleton Code", "Carleton State", "Sannio Code", "Sannio State", "*TestFul* Basic" }.

A	B_{cc}	B_{cs}	B_{sc}	B_{ss}	B_{tb}
TestFul basic	1.0000	1.0000	0.9987	0.9985	—
TestFul +du	1.0000	1.0000	0.9998	0.9998	1.0000
TestFul +puse	1.0000	1.0000	0.9998	0.9998	1.0000

With these last tests, we were also able to prune equivalent mutants and thus we also report the mutation score for non-equivalent mutants.

According to these data, *TestFul* outperforms the tests generated by the students, since its tests have both a higher average mutation score and a lower standard deviation. To have a statistical confirmation of this conjecture, we also performed a statistical hypothesis test between H_0 : "*tests created by* TestFul *outperform tests generated by human beings*" vs H_1 : "*tests created by* TestFul ***do not*** *outperform tests generated by human beings*", and the p-values[8] are reported in Table 4.7.

All the p-values are close to 1, hence we can safely assume that H_0 is true, and that the tests generated by *TestFul* are better than those created by human beings. Additionally, the last column of Table 4.7 compares the performance of the *basic* version of *TestFul* against the *+du* and *+puse* versions. Since the p-values are close to 1, there is a statistical confirmation that *TestFul* generates better tests if it also considers data-flow information.

[8]Fixed a significance level α, a p-value smaller or equal than α indicates that H_0 should be rejected. If α is set to 0.95, then results that are only 5% likely or less, given that H_0 is false, are deemed extraordinary.

Finally, as for costs, the tests generated by *TestFul* are (obviously) cheaper: its generation is completely automatic and it only requires 30 minutes (instead of the three or four hours given to the students). The cost for executing each generated test is also reasonable with an average of 7ms.

5. Related Work

The body of work on the automated generation of tests is vast [PY08]. During the years, researchers proposed several approaches to (semi-) automatically generate tests for a given piece of software. These work can differ for several reasons: the property of the system being tested (i.e., *functional* or *non-functional*), the scale of the system (i.e., *unit tests*, *integration tests*, *system tests*, or *acceptance tests*), the type of the system (e.g., *functions*, *classes*, or *agents*), the type and the amount of information of the system being used (i.e., *black-box*, *gray-box*, or *white-box*), or the technique used to generate tests (e.g., *symbolic execution* or *search-based approach*).

The goal of this chapter is not to be exhaustive, but to provide the reader a brief overview of the most promising approaches proposed in literature. For this reason, it focuses on those approaches that generate *functional unit tests*, grouping them into two groups: *Non Search-Based* and *Search-Based* approaches. The former are sketched in Section 5.1, which consider those work that do not rely on any search-based technique, and generate tests by analyzing the internal structure of the program or its specification. The latter are presented in Section 5.2, which deeply analyzes these approaches that reduces the generation of tests to a search problem, addressed using different search techniques.

5.1. Non Search-Based

The most promising "traditional" techniques that (semi-) automatically generate tests gravitate around two main groups: *specification-based* and *symbolic execution*.

5.1.1. Specification-Based

The generation of *functional unit tests* is a common and fundamental step in software development, but it presents several challenges. When produced manually, unit tests are often not sufficient to detect errors, mainly due to the limited amount of time available. However, even if some automated technique is used, there is the need of an *oracle*,

that is a technique to determine whether the system has behaved correctly or not. Albeit there are several techniques for *test oracles* [BY01], in recent years techniques based on a specification language (such as JML [LBR99, LCC+03]) received some attention. The presence of a machine-comprehensible specification can be exploited to generate tests. For example, Korat [BKM02] uses the method preconditions to automatically generate all non-isomorphic tests up to a fixed size. Then, it uses the postconditions to check if the method behaves correctly.

However, sometimes the specifications are partial or absent. Xie [Xie05, Xie06] tackled these cases, and proposed an approach that combines the specification inference [ECGN01] with the generation of the test. His work [PE05, XN05, XN06, CSX08] first infers an operational model of the system being tested. Then, he uses a classifier to identify *illegal*, *fault-revealing*, and *normal* inputs. *Illegal* inputs are just discarded, since they do not satisfy the system's preconditions. *Fault-revealing* inputs are reduced and prompted to the user, which can inspect them and decide if they reveal a fault or not. *Normal* inputs are used to generate new inputs, and continue the exploration.

5.1.2. Symbolic Execution

One of the most accepted approach is the *symbolic execution* [Kin75, Kin76, Cla76b, How77, RHC76].

Instead of supplying concrete inputs to the program (e.g., numbers), it provides symbolic values and uses them to execute the program (i.e., it performs a *symbolic execution*). When conditions are reached, it uses the symbolic values being compared to create a set of formulae —one for each branch— that contains the constraints to execute them. In this way, each path of the program can be associated with the set of constraints on the input parameter to traverse it [BEL75, Cla76a]. State of the art constraint solvers can be used to obtain the actual values for the parameters, and generate test data to reach all the paths of the program. Several approaches were proposed to apply these technique to object-oriented systems, and automatically generate tests for classes [BHR+00, BOP00, MOP02, Bal05, XMSN05, dPX+06, DRH07]. Moreover, Visser [VPP06] combines the symbolic execution with model checking to efficiently explore all the possible test sequences (up to a predefined size).

However, symbolic execution has some limitations. The number of paths of a program grows exponentially with the number of branches in the code, and it is often unbounded because of loop (this is known as the "path explosion" problem). Additionally, symbolic execution is limited

by the power of constraint solvers to deal with complex expressions. It also fails to manage large or complex programs, since maintaining and solving constraints becomes computationally intractable. To (partially) solve these problems, Larson and Austin [LA03] combined concrete and symbolic execution. The program is executed using values provided by the user, and symbolic path constraints are generated for that specific execution. Then, the collected constraints are solved to check whether there are potential inputs that —on the same execution path— would have led to a violation. Afterward, Godefroid [GKS05] extended the approach by proposing to incrementally generate test values. Instead of searching for violations, he uses the constraints to look for alternative paths. In particular, he explores all the paths of the program in a depth-first manner, by negating the conjunction of the path constraints. Sen [SMA05, SA06] goes a step further and developed a method to represent and track constraints that capture the behavior of a symbolic execution of a unit with memory graphs as inputs. Finally, these proposal were finalized and implemented in industrial-grade tools, such as *Java PathFinder* [PMB+08] and the *Visual Studio IDE* [TDH08].

5.2. Search-Based Approaches

Search Based Software Engineering (SBSE) applies search-based techniques to solve software-engineering problems. It suggests to reformulate the software engineering problem as an optimization problem. Accordingly, it is possible to leverage state of the art search algorithms and the high-performance computing power to efficiently find —from among all possible solutions to the problem— the one that meets the user's requirements. This process allows one to tackle problems often intractable by other methods, leads to innovative and insightful solutions, and partially or fully automates previously manually intensive tasks.

Search Based techniques have been applied to several software engineering problems, among which we mention Coding and Design Tools [JHH08, JGHL07, PHY10, NBY09, PEBC07, AWY08], Distribution and Maintenance [HT07, GHLM06, WCJP08, EB07, BE06, EB08], Management [WCTY09, GHA09, EB09, PHAQ07], Requirements / Specifications [ZH10, Zha10, ZAD+10, DZAN09, HKRY09, BHSS06, ZHM07, FHM+09, FHM+08, HSS06, ZFH08], and Testing / Debugging. The last category has been particularly active, and includes approaches to generate tests for verifying functional [Ham77, PHP99, Ton04, MTR08] and non-functional [GM02, BLS05, TWS06] properties of the system, to minimize the test suite [LOZ+07, HO09], and to automatically gener-

ate patches to fix the errors [WFGN10, FNWG09, WNGF09, FGFW10, SFW10].

In this section, we analyze search-based techniques to generate unit tests for verifying functional properties of the system. We organize them in two main groups: *blind* and *guided* search techniques.

5.2.1. Blind search

Random testing [Ham94] is probably the most famous search-based approach for the automatic generation of tests. It simply performs a random sequence of invocations on the system under test. Notwithstanding its simplicity, random testing can be as effective as other traditional approaches [CY96]. When failures are detected, random testing tools are able to provide witnesses, which are sequences of operations able to reveal them.

In contrast, the use of randomly-generated tests for regression testing is problematic. It is possible to identify two parts of the process: ensuring that errors fixed in the past are not reintroduced, and ensuring that the new version provides functionality that must be preserved. The former is satisfiable by means of witnesses. The latter instead necessitates to replay the whole sequence of random operations. This requires a huge amount of time, but allows one to obtain the same level of confidence as with the previous version of the application. To alleviate this issue it is possible to use the *random search*. Even if the two techniques share the same blind search strategy, random search requires an objective to maximize (e.g., the branch coverage). Accordingly, the execution of tests is monitored so to collect those tests that achieve the utmost level of the chosen objective.

Among available approaches, we mention AutoTest [MCLL07], one of the most advanced tools for random testing, specifically designed for object-oriented systems. AutoTest supports the evolution of each object through a sequence of random invocations of its methods and exploits contracts [Mey92] to spot if an output value, or the state of an object, is incorrect, and thus reveals a failure [LCO$^+$07]. Ciupa et al. [CLOM07] performed an empirical evaluation of random testing and found that the random seed influences achieved performances. We tried to investigate this dependency and we discovered that their tool uses a linear congruential method [Knu97] to generate random numbers —a weak method with severe limitations (e.g., the serial correlation between successive values) that is not recommended if randomness is critical. Accordingly, we replaced the random number generator of AutoTest with Mersenne Twister [MN98]. We applied the modified tool to a class implementing

a simple state machine, and performed several runs of 30 minutes (the same duration of the experiments reported in [CLOM07]), measuring the structural coverage. Figure 5.1 shows the coverage of the actual 100 def-use pairs. Even with a powerful random number generator, random

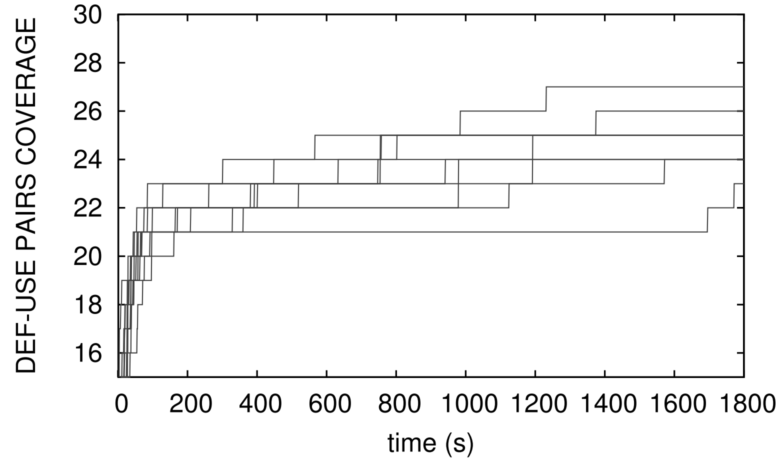


Figure 5.1.: Def-Use pairs coverage with random testing. Complete coverage consists of 100 def-use pairs.

testing cannot converge to the complete coverage of the 100 def-use pairs. These results are coherent with those in [CPL+08], which show that different executions of random tests tend to identify different failures. This phenomenon, easily explained by the fact that the search space is not properly explored even with long runs, is the major motivation of our work: a guided algorithm can explore the search space more fruitfully and thus lead to better tests.

To augment effectiveness, some works also propose *adaptive random testing* (ART) to ensure that generated values are equally distributed over the input domain [CLM04]. The idea is that the more distant values are, the better they are able to reveal failures. However, empirical studies show that ART tools do not discover failures earlier; instead they reveal a different set.

Random testing techniques requires to tune some parameters, so to achieve an high fault-effectiveness. Albeit some default parameters has been proposed [CPL+08], the optimal setting depends on the class being tested. For this reason, Andrews [AHLL06] employs an evolutionary algorithm to find the optimal settings for the random testing session. The evolutionary algorithm uses, as fitness function to determine the quality of a given setting, the level of structural coverage that a short

session of random testing achieves.

Other approaches, such as [PLEB07], enhance traditional random testing with taboo-search. Taboo-search techniques explore the search space in an incremental manner, as they consider the neighborhood of the solution found so far. Additionally, they memorize the last decisions made, so to steer the search process towards new directions and avoid to be trapped in local optima. Accordingly, these approaches generate tests incrementally by adding a randomly-chosen operation to a previous test. They analyze each test, and categorize it as *error-revealing*, *new*, or *illegal*. Error-revealing tests are prompted to the user, and they are not used to create new tests anymore. New tests are able to create objects not equivalent to those created by previous tests. They are presented to the user for regression testing (they represent a normal behavior of the system), and are used as basis for generating new tests. Illegal tests contain an illegal operation that violates the invoked method's preconditions. These tests are discarded and are not used for generating new tests.

5.2.2. Guided search

Other search-based test generation techniques are guided since the search process is directed towards the satisfaction of a goal.

In structural testing, the goal is to reach the maximum coverage for a given criterion (e.g., cover all branches in the system). McMinn [McM04] highlights two main approaches: coverage-oriented and structure-oriented. The former rewards the tests that cover more structural elements; for example, Watkins [Wat95] tries to achieve full path coverage on stateless systems. Its work was not able to achieve good results, mainly because the approach was not able to provide enough guidance to the search process.

Structure-oriented approaches tackle separately each uncovered structural element identified by the coverage criterion [Kor90]. Before applying the search algorithm, the system under test is analyzed to select the set of structural elements of interest. For example, branch coverage identifies all the edges outgoing from each conditional statement, and path coverage identifies all the execution paths of the system. Then, the algorithm selects one of these structural elements and uses a search algorithm to generate a test able to reach it. These proposals follow the "divide and conquer" approach, handling structural elements separately even if they are often tightly related. Trying to reach each element by starting from scratch requires unnecessary effort. This phenomenon is particularly important in stateful systems since a lot of effort is required

to put objects in useful states.

There are several works that refine the structure-oriented approach by proposing functions able to better guide the search process. Initially, they use the control flow graph of the program to judge the distance between the test and the desired structural element [MMS01, WBS01, BSS02, Arc10a]. By comparing the execution flow of the test with the control flow graph, one can identify the conditional statement responsible for the deviation of the execution flow from the target. The quality of the test is then judged by using two elements: *approach level* and *branch distance*. The first measures the number of conditional statements between the flow deviation and the target. The second focuses on the conditional statements where the execution flow deviates from the desired one, and measures the distance between the actual values and those needed to take the branch that would lead to the target.

These proposals work on stateless systems: they consider the invocation of a single function and generate the input parameters to reach the selected structural element. In contrast, Tonella [Ton04] was the first to focus on object-oriented systems. For each branch in the class under test, he searches for a sequence of operations able to prepare the state of objects and exercise the selected branch. However, his work does not capitalize on the state of objects: the fitness function is the same as that of works that operate on stateless systems, and when a new branch is targeted, the search process restarts from the beginning.

Arcuri's research [AY07c] was set on the generation of tests for object-oriented systems. He particularly stressed containers [AY07a, AY08], which represent an interesting type of classes. His proposal integrates the Tonella's work with a more advanced fitness function, which consider both the approach level and the branch distance. However, Arcuri's approach systematically applies the *divide and conquer* paradigm, does not capitalize on the state of objects, and the search process starts from scratch every time.

The proposals presented so far are able to successfully guide the search process towards the selected target, but they are not able to deal with dependencies not explicit in the control flow graph. Even if some attempts to handle common cases were made, stateful systems introduce many more hidden dependencies, and to the best of our knowledge, nobody has proposed a fitness function able to make them explicit.

For example, Cheon [CK06] leverages JML specifications to provide guidance to the evolutionary algorithm. However, his proposal has severe limitations, and a high-quality specification of classes often is missing.

Wappler [WS07] considers encapsulation of class member which prevents methods from being freely accessed. If the target is contained in a

non-public method, the test must firstly invoke such method, and later it is possible to use the traditional fitness function. Accordingly, they propose to modify the fitness function so to reckon the call points, allowing the non-public method to be called.

The “flag variable” is another typical example that these proposals fail to handle. In this case, the information gathered from the control flow graph is misleading, and it might result in a coarse guidance. Instead, one should integrate the control-flow analysis with the data-flow information. Ferguson was the first who proposed the chaining approach [FK96], to combine control-flow with data-flow analysis. His approach was refined by many other authors [BS03, MH03, MH04, MH05, MH06, AAA09].

Harman [HHH+04, MBH09] was the first to introduce *Testability Transformation*. It is a source-to-source transformation that is applied on the system being tested, with the purpose to ease the application of test data generation technique and improve their performance. For example, one transformation modifies the program so to remove the use of flag variables. The modified version of the program contains hence a very complicated control-flow structure that mime the same behavior of the original program, but it is better suited for the automated test generation. The modified program is to be used for the test-data generation only, as it is more complex to understand and less maintainable. Additionally, some transformations —in order to ease the data generation for some targets— might also change the semantics of other parts of the program. Interestingly, the application of testability transformations open new research possibilities. Tonella [MHBT06] transforms the program so to explicit the different paths that lead to the target and makes several sub-population (species) compete over them. McMinn [McM09] uses testability transformations to create a de-optimized version of the program to use as oracle. For example, they note that the usage of some floating-point variables might lead to approximation errors, avoidable by using more advanced (but more expensive) representations. The proposal follows by monitoring the differences between the base version of the program and the de-optimized one. He uses an evolutionary algorithm to maximize the difference and allow the user to understand the importance of the error.

There are several proposals that apply different search techniques to solve the generation of tests. Arcuri experiments novel evolutionary search methods, namely Memetic Algorithms [AY07b] and Estimation of Distribution Algorithms [SSAY07], achieving promising results. Windisch et al. [WWW07] successfully applies Particle Swarm Optimization to software testing, which —in his experiments— outperforms evolutionary algorithms. Wappler [WW06b, WW06a] uses the strongly-

typed genetic programming to generate tests. His approach relies on a tree-based representation of tests, which ensures the feasibility of all operations. Additionally, he divides the search process into two steps. The first one concentrates on the structure of the test, working on the tree-based representation. In this phase, the test is a skeleton that pinpoints the methods to call, but without specifying the actual parameters. These are the subject of the second step of the search process, where a genetic algorithm concentrates on finding the optimal values (both primitive types and objects) for parameters. A similar search strategy is also used in other works. For example, [LI08, LI07] leverages a finite state machine to generate the sequence of method calls, and then use a genetic algorithm to find the optimal value for actual parameters.

Some other approaches use genetic algorithms to generate tests for data-flow coverage. For example, Ghiduk et al. [GHG07] target du-pairs separately: for each pair, they provide a fitness function that targets a complex constraint: the definition and use must be executed without any kill in between. To provide a smoother guidance, they propose a measure of closeness derived from the dominance relations. However, this work has the same limitations as Tonella's work: it does not capitalize when an object reaches a valuable state and tackles each du-pair separately.

Harman and McMinn [HM07] were the first to perform a theoretical study on the scenarios in which evolutionary algorithms are suitable for structural test case generation. They identify a particular type of fitness function, namely the *Royal Road* [MFH91], where evolutionary algorithms provably perform well. Harman et al. [HHL+07] studied the influence between the size of the input domain (the search space) and the performance of search-based algorithms. They discovered that, finding input parameters that do not influence a target, helps guided methods but does not help blind methods. Other authors continue this theoretical analysis. Arcuri concentrates on why meta-heuristics can be effective in software testing [Arc09], and then he considers the role of the test's length on the ability to reach all the targets of the class [Arc10c].

Finally, there are approaches that aims to generate tests by using mutation analysis [LLR07, ABA07, MMT03, BFJT05]. The authors claim that other coverage criteria might generate tests with a low fault detection effectiveness. Instead, mutant analysis mime errors that real developers do. It correctly calculates the quality of tests by counting the number of mutants they kill. Since mutant analysis is very expensive to use, they use a search technique inspired to the immune system. Mutants represents infections, while tests are the vaccine. Both of them are allowed to evolve and compete, developing more subtle mutations and more sophisticated tests.

6. Conclusions and Future Research Directions

The research presented in this dissertation proposes *TestFul*, that is a novel approach for the automatic unit test generation tailored to stateful systems. In particular, we tackle the test-generation problem as a search problem, solved using a holistic evolutionary algorithm. Compared to the state of the art in search-based test generation, *TestFul* recognizes and reuses useful state configurations to exercise different features.

To augment the efficiency of the approach, we leverage three techniques: *local search*, *seeding*, and *fitness inheritance*. *Local search* integrates the global evolutionary search to form a hybrid approach. We establish a synergic cooperation between the two searches, as the results of one are used as tarting point of the other. *Seeding* provides *TestFul* with a high-quality initial population that can speed up the initial part of the evolutionary process. To this end, we both (i) run a short run of random test, and (ii) we adapt tests already generated for compatible classes. *Fitness Inheritance* speeds up the search process by replacing the evaluation of the fitness function of some individuals with an estimated fitness inherited from their parents.

TestFul leverages complementary coverage criteria to drive the evolution of tests. Accordingly, it employs a multi-objective evolutionary algorithm to evolve candidate tests by combining the different coverage criteria with the compactness of the tests.

Several complementary coverage criteria are in charge to measure the quality of tests and guide the test-generation process. The *behavioral coverage* criterion employs a black-box analysis technique to detect the behaviors of the class that tests execute, and reward tests accordingly. The *statement coverage* and *branch coverage* criteria are white-box: both measure the quality of tests according to the coverage the control-flow graph. These criteria are able to provide an effective guidance towards their fulfillment, hence most of the test-generation approaches uses them. However, those approaches only use this information to judge tests, but it is known that tests with an high level of branch coverage might not detect several errors. For this reason, we make *TestFul* to consider data-flow information in its test generation process. In this way, it can detect

data-dependencies among methods and combine this information with control-flow data. Accordingly, we reward tests also by considering both *all du-pairs* coverage and *all p-uses* coverage criteria.

To validate the effectiveness of each proposal, we devised a benchmark composed of more than 200 classes, taken from literature, real-world applications, and other independent benchmarks. The comparison is performed by both considering (i) previous versions of *TestFul*, (ii) other search-based approaches, and (iii) more "traditional" ways to generate tests. Obtained results, although limited to the considered problems, highlight the validity of the overall proposal, and show its good performances. It seems that other approaches have a limited capability of enabling complex behaviors of the class under test, due to some inherent limits of the symbolic execution or a poor guidance for search-based approaches. In contrast, *TestFul* explores the search space by using an incremental approach (i.e., by exploring the space closer to current solutions). This allows *TestFul* to put objects in useful states and therefore it typically pays on classes with complex states.

Future Research Directions

The novel approach introduced in this research opens interesting research directions:

- We show the ability of the behavioral coverage to guide *TestFul* and generate tests with a high fault-detection ability. However, the behavioral coverage requires the user to provide additional information. This information is used both to discern meaningful properties of the system from auxiliary ones, and to abstract them so to elicit the behavioral model of the system. It would be interesting to automate also this step, and provide a way to automatically detect how to reassemble the behavioral model.

- The data-flow coverage criteria are able to detect methods that cooperate by exchanging data. We already show how to apply the local search when a p-use is involved, and we achieved important results. However, it would be interesting to consider a generic def-use pair not yet exercised, and guide the evolution towards its fulfillment.

- Even if the chosen criteria assign each test with a good quality, it would be interesting to consider if other analysis techniques provide a better insight on the system under test.

- We moved some early steps on the exploitation of the domain knowledge to improve the search strategy. It would be interesting to continue this research and systematically perform the *residual testing* [PY99]. Moreover, it would be interesting to reuse both the information present in user-written tests and the "normal" execution of the system, and perform the

- The proposal presented in this dissertation focuses exclusively on the unit-testing. It would be interesting to consider coarse-grained levels, and focus on the integration testing.

- Even if the approach we present aspire to applicable to every stateful systems, currently we only focused on object-oriented programs. It would be interesting to consider other types of stateful systems, namely components and services.

- Beside the local search, the evolutionary algorithm modifies tests by performing a random change, which uses a uniform selection mechanism. It is interesting to move a step further, and improve both (i) the selection of the objects from the repository and (ii) the selection of the statements to apply to the objects — two key factors for the development of tests in classes with complex internal states. For this purpose, one should to investigate the use of a multi-objective probabilistic model-building genetic algorithm to replace the current evolutionary engine. The probabilistic model should be mainly used to implement an adaptive mutation (and not for crossover, as in the typical "Probabilistic Model Building Genetic Algorithm"); thus, it will estimate the probability to add/replace a certain object to the test sequence or to add/replace a certain type of statement.

A. Mutation Testing

Mutation analysis [DLS78, ABLN06] is a widely-used technique to determine the effectiveness in detecting faults of the different tests. It uses mutant operators to generate multiple mutated version of the class, each one with a single fault seeded.

Offut [OLR$^+$96] analyzed the different mutant operators and found that only five of them are sufficient for mutation analysis. Consequently, we created mutants of the original program by using these five operators:

- *Absolute Value Insertion* focuses on arithmetic expressions, forcing the result to take zero, a positive value, or a negative one;
- *Arithmetic Operator Replacement* focuses on arithmetic expressions, replacing arithmetic operators (e.g., +) with others;
- *Logical Connector Replacement* focuses boolean expressions, replacing logical operators (e.g., &&) with others, or making the whole formula true or false;
- *Relational Operator Replacement* focuses on comparisons, replacing relational operators (e.g., $<$) with others;
- *Unary Operator Insertion* focuses on expressions, replacing each used numerical value v with 0, $-v$, $v+1$, or $v-1$, and each boolean value with $true$ or $false$;

The application of such mutant operators might create *equivalent mutants*, which are semantically identical to the original program, hence tests cannot kill them. For example, $a > b$ and $a \geq b$ are equivalent if a cannot be equal to b. To correctly judge the fault-detection ability of tests, one should (manually) identify and prune equivalent mutants. The detection of equivalent mutants is extremely time-consuming, since it must be performed manually. In this work, we prune equivalent mutants by using a commonly used heuristic that marks mutants not killed by any test as equivalent mutants [BS79, DLS78].

We generated tests for each class using the different configurations of *TestFul*. Each time, the tool was given 20 minutes of CPU-time on an Intel Xeon E5530@2.40GHz with 6 gigabytes of RAM. To achieve

more accurate results, we repeated each experiment ten times, and the comparison we made is on the mean values. A test is run on each of these versions, and if a failure manifests itself, then the mutant is said to be *killed*; otherwise the mutant is *alive*. The fault detection effectiveness of each test is measured by considering the percentage of killed mutants.

To recognize if a mutant is killed, we analyzed wether the mutated program behaves differently from the original one. Consequently, we analyzed the test execution on the original program, and after each invocation we observed the status of the object accepting the method call and of produced result. Then, we ran the mutated versions of the program, and we monitored again the same properties: differences between the normal execution and the mutated ones witnessed the capability of the test to reveal the effects of the mutator and consequently the mutant was killed. Moreover, some mutants might lead the program not to terminate (for example, if we set to zero the step of a loop). Obviously, a test able to make these mutated programs not to terminate is able to kill these mutants, since the observed behavior is different from the original one. Consequently, we set a threshold for the test executions on mutants, calculated as five times the execution time of the tests on the original program. If a mutant made a test require more than this threshold to execute, than we marked it as *killed*.

Bibliography

[AAA09] Zeina Awedikian, Kamel Ayari, and Giuliano Antoniol. Mc/dc automatic test input data generation. In Rothlauf [Rot09], pages 1657–1664.

[ABA07] Kamel Ayari, Salah Bouktif, and Giuliano Antoniol. Automatic mutation test input data generation via ant colony. In Lipson [Lip07], pages 1074–1081.

[ABL05] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 402–411. ACM, 2005.

[ABLN06] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE TSE*, 32(8):608–624, 2006.

[AHLL06] James H. Andrews, Susmita Haldar, Yong Lei, and Felix Chun Hang Li. Tool support for randomized unit testing. In Mayer and Merkel [MM06], pages 36–45.

[Apa] Apache Software Foundation. The Apache Commons Mathematics Library. `http://commons.apache.org/math/`.

[Arc09] Andrea Arcuri. Insight knowledge in search based software testing. In Rothlauf [Rot09], pages 1649–1656.

[Arc10a] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In ICST2010 [ICS10], pages 205–214.

[Arc10b] Andrea Arcuri. Longer is Better: On the Role of Test Sequence Length in Software Testing. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[Arc10c] Andrea Arcuri. Longer is better: On the role of test sequence length in software testing. In ICST2010 [ICS10], pages 469–478.

[AWY08] Andrea Arcuri, David Robert White, , and Xin Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *Proceedings of the 7th International Conference on Simulated Evolution And Learning (SEAL '08)*, pages 61–70, Melbourne, Australia, 7-10 December 2008. Springer.

[AY07a] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 2007.

[AY07b] Andrea Arcuri and Xin Yao. A memetic algorithm for test data generation of object-oriented software. In Xin Yao, editor, *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 2048–2055, 2007.

[AY07c] Andrea Arcuri and Xin Yao. On test data generation of object-oriented software. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 72–76, Sept. 2007.

[AY08] Andrea Arcuri and Xin Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075 – 3095, 2008. Nature Inspired Problem-Solving.

[Bal05] T. Ball. A theory of predicate-complete test coverage and generation. In *Formal Methods for Components and Objects*, pages 1–22. Springer, 2005.

[BE06] Iain Bate and Paul Emberson. Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems. In *Proceedings of the 12th IEEE Real-Time And Embedded Technology And Applications Symposium (RTAS '06)*, pages 221–230, San Jose, California, USA, 4-7 April 2006. IEEE.

[Bei90] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[BEL75] R.S. Boyer, B. Elspas, and K.N. Levitt. SELECT — A formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245. ACM, 1975.

[BFJT05] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Softw. Test., Verif. Reliab.*, 15(2):73–96, 2005.

[BHR+00] Thomas Ball, Daniel Hoffman, Frank Ruskey, Richard Webber, and Lee J. White. State generation and automated class testing. *Softw. Test., Verif. Reliab.*, 10(3):149–170, 2000.

[BHSS06] Paul Baker, Mark Harman, Kathleen SteinhÃűfel, and Alexandros Skaliotis. Search based approaches to component selection and prioritization for the next release problem. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 176–185, Philadelphia, Pennsylvania, 24-27 September 2006. IEEE.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *ISSTA*, pages 123–133, 2002.

[BLS05] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. In Beyer and O'Reilly [BO05], pages 1021–1028.

[BO05] Hans-Georg Beyer and Una-May O'Reilly, editors. *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005.* ACM, 2005.

[BOP00] Ugo A. Buy, Alessandro Orso, and Mauro Pezzè. Automated Testing of Classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 39–48, 2000.

[BS79] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Technical Report 276, Yale University, Department of Computer Science, 1979.

[BS03] André Baresel and Harmen Sthamer. Evolutionary testing of flag conditions. In Cantú-Paz et al. [CPFD+03], pages 2442–2454.

[BSS02] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In William B. Langdon, Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant Honavar, Gunter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska, editors, *GECCO*, pages 1329–1336. Morgan Kaufmann, 2002.

[BY01] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Department of Computer and Information Science, 2001.

[Cat06] Mike Cattolico, editor. *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006*. ACM, 2006.

[CK06] Yoonsik Cheon and Myoung Kim. A specification-based fitness function for evolutionary testing of object-oriented programs. In Cattolico [Cat06], pages 1953–1954.

[Cla76a] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[Cla76b] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM.

[CLM04] T.Y. Chen, H. Leung, and I.K. Mak. Adaptive Random Testing. In Springer, editor, *Proceedings of Advances in Computer Science — ASIAN*, volume 3321/2005, 2004.

[CLOM07] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental Assessment of Random Testing for Object-Oriented Software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 84–94, 2007.

[cob] Cobertura. `http://cobertura.sourceforge.net/`.

[CPFD+03] Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar

Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasa Jonoska, and Julian F. Miller, editors. *Genetic and Evolutionary Computation - GECCO 2003, Genetic and Evolutionary Computation Conference, Chicago, IL, USA, July 12-16, 2003. Proceedings, Part II*, volume 2724 of *Lecture Notes in Computer Science*. Springer, 2003.

[CPL+08] Ilinca Ciupa, Alexander Pretschner, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. On the predictability of random tests for object-oriented software. In ICST2008 [ICS08], pages 72–81.

[CS04] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[CSX08] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–37, 2008.

[CY96] Tsong Yueh Chen and Yuen-Tak Yu. On the expected number of failures detected by subdomain testing and random testing. *IEEE TSE*, 22(2):109–119, 1996.

[DAPM02] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[DER05] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the praticing programmer. *IEEE Computer*, 11(4):34–41, Apr 1978.

[DLWZ06] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining Object Behavior with ADABU.

In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24, New York, NY, USA, 2006. ACM.

[dPX+06] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE*, pages 59–68. IEEE Computer Society, 2006.

[DRH07] Xianghua Deng, Robby, and John Hatcliff. Robby, and John Hatcliff. Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-Oriented Systems. In *Proc. Testing: Academia & Industry Conference–Practice & Research Techniques*, pages 3–12, 2007.

[DZAN09] Juan J. Durillo, Yuanyuan Zhang, Enrique Alba, and Antonio J. Nebro. A study of the multi-objective next release problem. In *Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE '09)*, pages 49–58, Cumberland Lodge, Windsor, UK, 13-15 May 2009. IEEE.

[EB07] Paul Emberson and Iain Bate. Minimising task migration and priority changes in mode transitions. In *Proceedings of the 13th IEEE Real-Time And Embedded Technology And Applications Symposium (RTAS '07)*, pages 158–167, Bellevue, Washington, USA, 3-6 April 2007. IEEE.

[EB08] Paul Emberson and Iain Bate. Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems. In *Proceedings of the 2008 Real-Time Systems Symposium (RTSS '08)*, pages 270–279, Barcelona, Spain, 30 November - 3 December 2008. IEEE.

[EB09] Paul Emberson and Iain Bate. Stressing search with scenarios for flexible solutions to real-time task allocation problems. *IEEE Transactions on Software Engineering*, 2009.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 213–224, 1999.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, 2001.

[FGFW10] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In Martin Pelikan and Jürgen Branke, editors, *GECCO*, pages 965–972. ACM, 2010.

[FHM+08] Anthony Finkelstein, Mark Harman, S. Afshin Mansouri, Jian Ren, and Yuanyuan Zhang. “fairness analysis" in requirements assignments. In *Proceedings of the 16th IEEE International Requirements Engineering Conference (RE '08)*, pages 115–124, Barcelona, Catalunya, Spain, 8-12 September 2008. IEEE.

[FHM+09] Anthony Finkelstein, Mark Harman, S. Afshin Mansouri, Jian Ren, and Yuanyuan Zhang. A search based approach to fairness analysis in requirement assignments to aid negotiation, mediation and decision making. *Requirements Engineering Journal (RE '08 Special Issue)*, 14(4):231–245, December 2009.

[FK96] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM TOSEM*, 5(1):63–86, 1996.

[FNWG09] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In Rothlauf [Rot09], pages 947–954.

[GHA09] Stefan Gueorguiev, Mark Harman, and Giuliano Antoniol. Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09)*, pages 1673–1680 (Best Paper Award), MontrÃľal, Canada, 8-12 July 2009. ACM.

[GHG07] Ahmed S. Ghiduk, Mary Jean Harrold, and Moheb R. Girgis. Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC)*, pages 41–48, 2007.

[GHLM06] Nicolas Gold, Mark Harman, Zheng Li, and Kiarash Mahdavi. Allowing overlapping boundaries in source code using a search based approach to concept binding. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 310–319, Philadelphia, USA, 24-27 September 2006. IEEE.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[GM02] Hans-Gerhard Groß and Nikolas Mayer. Evolutionary testing in component-based real-time system construction. In Erick Cantú-Paz, editor, *GECCO Late Breaking Papers*, pages 207–214. AAAI, 2002.

[Gol02] David E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publisher, 2002.

[Ham77] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE TSE*, 3(4):279–290, 1977.

[Ham94] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[Ham06] Dick Hamlet. When only random testing will do. In Mayer and Merkel [MM06], pages 1–9.

[HFGO94] Monica Hutchins, Herbert Foster, Tarak Goradia, and Thomas J. Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 191–200, 1994.

[HHH+04] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE TSE*, 30(1):3–16, 2004.

[HHL+07] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In Ivica Crnkovic and Antonia Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 155–164. ACM, 2007.

[HKRY09] Mark Harman, Jens Krinke, Jian Ren, and Shin Yoo. Search based data sensitivity analysis applied to requirement engineering. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09)*, pages 1681–1688, Montreal, Canada, 8-12 July 2009. ACM.

[HM07] Mark Harman and Phil McMinn. A theoretical & empirical znalysis of evolutionary testing and hill climbing for structural test data generation. In David S. Rosenblum and Sebastian G. Elbaum, editors, *ISSTA*, pages 73–83. ACM, 2007.

[HO09] Hwa-You Hsu and A. Orso. Mints: A general framework and tool for supporting test-suite minimization. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 419 –429, May 2009.

[How77] W.E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3:266–278, 1977.

[HSS06] Mark Harman, Alexandros Skaliotis, and Kathleen SteinhÃűfel. Search-based approaches to the component selection and prioritization problem. In *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, pages 1951–1952, Seattle, Washington, USA, 8-12 July 2006. ACM.

[HT07] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO '07)*, pages 1106–1113, London, England, 7-11 July 2007. ACM.

[ICS08] *First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008*. IEEE Computer Society, 2008.

[ICS10] *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. IEEE Computer Society, 2010.

[JGHL07] Tao Jiang, Nicolas Gold, Mark Harman, and Zheng Li. Locating dependence structures using search-based slicing.

Information and Software Technology, 50(12):1189–1209, November 2007.

[JGr] Jgrapht. `http://www.jgrapht.org/`.

[JHH08] Tao Jiang, Mark Harman, and Youssef Hassoun. Analysis of procedure splitability. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08)*, pages 247–256, Antwerp, Belgium, 15-18 October 2008. IEEE.

[jun] JUnit. `http://www.junit.org/`.

[KGH+95] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Chris Chen, Young-Si Kim, and Young-Kee Song. Developing an object-oriented software testing and maintenance environment. *Commun. ACM*, 38(10):75–87, 1995.

[Kin75] J. King. A new approach to program testing. *Programming Methodology*, pages 278–290, 1975.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KKS98] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230 –239, jun. 1998.

[Knu97] Donald Ervin Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms, chapter 3.2.1: The Linear Congruential Method, pages 10–26. Addison-Wesley, 1997.

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE TSE*, 16(8):870–879, 1990.

[LA03] Eric Larson and Todd Austin. High coverage detection of input-related security facults. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[LBR99] G. Leavens, A. Baker, and C. Ruby. JML: A notation for detailed design. *KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE*, pages 175–188, 1999.

[LCC+03] G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D.R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In *Formal Methods for Components and Objects*, pages 262–284. Springer, 2003.

[LCO+07] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract Driven Development = Test Driven Development - Writing Test Cases. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 425–434, 2007.

[LI07] R. Lefticaru and F. Ipate. Automatic state-based test generation using genetic algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing, 2007. SYNASC. International Symposium on*, pages 188 –195, sept. 2007.

[LI08] R. Lefticaru and F. Ipate. Search-based testing using state-based fitness. In *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, page 210, april 2008.

[Lip07] Hod Lipson, editor. *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*. ACM, 2007.

[LLR07] Konstantinos Liaskos, Konstantinos Liaskos, and Marc Roper. Automatic test-data generation: An immunological approach. In Marc Roper, editor, *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 77–81, 2007.

[LOZ+07] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In Stirewalt et al. [SEF07], pages 417–420.

[LP97] W. B. Langdon and R. Poli. Fitness causes bloat. In *Second On-Line World Conference on Soft Computing in Engineering Design and Manufacturin*, pages 13–22, June 1997.

[MBH09] Phil McMinn, David Binkley, and Mark Harman. Empirical evaluation of a nesting testability transformation for evolu-

tionary testing. *ACM Trans. Softw. Eng. Methodol.*, 18(3), 2009.

[MBL10] Matteo Miraz, Luciano Baresi, and Pier Luca Lanzi. TestFul: an Evolutionary Test Approach for Java. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[MBLD10] Samar Mouchawrab, Lionel C. Briand, Yvan Labiche, and Massimiliano Di Penta. Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010.

[MCLL07] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Liu. Automatic Testing of Object-Oriented Software. In *Proceedings of SOFSEM 2007: 33rd Conference on Current Trends in Theory and Practice of Computer Science*, pages 114–129. Springer, 2007.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.

[McM09] Phil McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In Rothlauf [Rot09], pages 1689–1696.

[Mey92] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, Oct 1992.

[MFH91] Melanie Mitchell, Stephanie Forrest, and John H. Holland. The royal road for genetic algorithms: Fitness landscapes and ga performance. 1991.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[MH03] Phil McMinn and Mike Holcombe. The state problem for evolutionary testing. In Cantú-Paz et al. [CPFD+03], pages 2488–2498.

[MH04] Phil McMinn and Mike Holcombe. Hybridizing evolutionary testing with the chaining approach. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund K. Burke, Paul J. Darwen, Dipankar Dasgupta, Dario

Floreano, James A. Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andrew M. Tyrrell, editors, *GECCO (2)*, volume 3103 of *Lecture Notes in Computer Science*, pages 1363–1374. Springer, 2004.

[MH05] Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In Beyer and O'Reilly [BO05], pages 1013–1020.

[MH06] Phil McMinn and Mike Holcombe. Evolutionary testing using an extended chaining approach. *Evolutionary Computation*, 14(1):41–64, 2006.

[MHBT06] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to searchbased test data generation. In Pollock and Pezzè [PP06], pages 13–24.

[MLB09] Matteo Miraz, Pier Luca Lanzi, and Luciano Baresi. Test-Ful: Using a Hybrid Evolutionary Algorithm for Testing Stateful Systems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1947–1948, 2009.

[MLB10] Matteo Miraz, Pier Luca Lanzi, and Luciano Baresi. Improving Evolutionary Testing by Means of Efficiency Enhancement Techniques. In *in Proceedings of the IEEE Congress on Evolutionary Computation*, 2010.

[MM06] Johannes Mayer and Robert G. Merkel, editors. *Proceedings of the 1st International Workshop on Random Testing, RT 2006, Portland, Maine, July 20, 2006.* ACM, 2006.

[MMS01] Christoph C. Michael, Gary McGraw, and Michael Schatz. Generating software test data by evolution. *IEEE TSE*, 27(12):1085–1110, 2001.

[MMT03] Peter May, Keith Mander, and Jon Timmis. Software vaccination: An artificial immune system approach to mutation testing. In *Artificial Immune Systems*, volume 2787 of *Lecture Notes in Computer Science*, pages 81–92. Springer Berlin / Heidelberg, 2003.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform

pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[MOP02] V. Martena, A. Orso, and M. Pezze. Interclass testing of object oriented software. In *Engineering of Complex Computer Systems, 2002. Proceedings. Eighth IEEE International Conference on*, pages 135–144, 2002.

[MS07] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 416–426, 2007.

[MTR08] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of ajax web applications. In ICST2008 [ICS08], pages 121–130.

[NBY09] Vivek Nallur, Rami Bahsoon, and Xin Yao. Self-optimizing architecture for ensuring quality attributes in the cloud. In *Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA '09)*, Cambridge, UK, 14-17 September 2009.

[OLR+96] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM TOSEM*, 5(2):99–118, 1996.

[PE05] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 504–527. Springer, 2005.

[PEBC07] Simon Poulding, Paul Emberson, Iain Bate, and John A. Clark. An efficient experimental methodology for configuring search-based design algorithms. In *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium (HASE '07)*, pages 53–62, Dallas, Texas, USA, 14-16 November 2007. IEEE.

[PHAQ07] Massimiliano Di Penta, Mark Harman, Giuliano Antoniol, and Fahim Qureshi. The effect of communication overhead on software maintenance project staffing: a search-based approach. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM '07)*, pages 315–324, Paris, France, 2-5 October 2007. IEEE.

[PHP99] Roy P. Pargas, Mary Jean Harrold, and Robert Peck. Test-data generation using genetic algorithms. *Softw. Test., Verif. Reliab.*, 9(4):263–282, 1999.

[PHY10] Kata Praditwong, Mark Harman, and Xin Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 2010.

[PLB08] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding Errors in .NET with Feedback-Directed Random Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 87–96, 2008.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[PMB+08] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing NASA Software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 15–26, 2008.

[PP06] Lori L. Pollock and Mauro Pezzè, editors. *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*. ACM, 2006.

[PY99] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *ICSE*, pages 277–284, 1999.

[PY04] Mauro Pezzè and Michal Young. Testing object oriented software. In *ICSE*, pages 739–740, 2004.

[PY08] Mauro Pezzè and Michal Young. *Software testing and analysis*. Wiley, 2008.

[RHC76] C.V. Ramamoorthy, S.-B.F. Ho, and W.T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2:293–300, 1976.

[roo] Roops: Benchmarks for reachability in object-oriented programs. `http://code.google.com/p/roops/`.

[Rot09] Franz Rothlauf, editor. *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*. ACM, 2009.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE TSE*, 11(4):367–375, 1985.

[SA06] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006.

[Sas02] Kumara Sastry. Evaluation-Relaxation Schemes for Genetic and Evolutionary Algorithms. Technical Report 2002004, University of Illinois at Urbana-Champaign, Urbana, IL, February 2002.

[Sas07] Kumara Sastry. *Genetic algorithms and genetic programming for multiscale modeling: Applications in materials science and chemistry and advances in scalability*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2007.

[SEF07] R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors. *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. ACM, 2007.

[SFW10] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 313–316. ACM, 2010.

[SMA05] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272. ACM, 2005.

[SSAY07] Ramon Sagarna, Ramon Sagarna, Andrea Arcuri, and Xin Yao. Estimation of distribution algorithms for testing object oriented software. In Andrea Arcuri, editor, *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 438–444, 2007.

[Tas02] G. Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology RTI Project, 2002.

[TDH08] N. Tillmann and J. De Halleux. Pex: white box test generation for. NET. In *Proceedings of the 2nd international conference on Tests and proofs*, pages 134–153. Springer-Verlag, 2008.

[Ton04] Paolo Tonella. Evolutionary Testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[TWS06] Marouane Tlili, Stefan Wappler, and Harmen Sthamer. Improving evolutionary real-time testing. In Cattolico [Cat06], pages 1917–1924.

[UOH93] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. In *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '93, pages 139–148, New York, NY, USA, 1993. ACM.

[vim] Vim. `http://www.vim.org`.

[VPP06] Willem Visser, Corina S. Pasareanu, and Radek Pelánek. Test input generation for java containers using state matching. In Pollock and Pezzè [PP06], pages 37–48.

[Wat95] A. Watkins. The automatic generation of test data using genetic algorithms. In *In Proceedings of the Fourth Software Quality Converence*, pages 300–309, 1995.

[WBS01] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14):841–854, 2001.

[WCJP08] David R. White, John A. Clark, Jeremy Jacob, and Simon M. Poulding. Searching for resource-efficient programs: Low-power pseudorandom number generators. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO '08)*, pages 1775–1782, Atlanta, GA, USA, 12-16 July 2008. ACM.

[WCTY09] Zai Wang, Tianshi Chen, Ke Tang, and Xin Yao. A multi-objective approach to redundancy allocation problem in parallel-series systems. In *Proceedings of the 10th IEEE Congress on Evolutionary Computation (CEC '09)*, pages 582–589, Trondheim, Norway, 18-21 May 2009. IEEE.

[WFGN10] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5):109–116, 2010.

[WNGF09] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374. IEEE, 2009.

[WRH+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[WS07] Stefan Wappler and Ina Schieferdecker. Improving evolutionary class testing in the presence of non-public methods. In Stirewalt et al. [SEF07], pages 381–384.

[WW06a] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 851 –858, 2006.

[WW06b] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In Cattolico [Cat06], pages 1925–1932.

[WWW07] Andreas Windisch, Stefan Wappler, and Joachim Wegener. Applying particle swarm optimization to software testing. In Lipson [Lip07], pages 1121–1128.

[Xie05] Tao Xie. *Improving Effectiveness of Automated Software Testing in the Absence of Specifications.* PhD thesis, University of Washington, 2005.

[Xie06] Tao Xie. Improving Effectiveness of Automated Software Testing in the Absence of Specifications. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM)*, pages 355–359, 2006.

[XMSN05] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 365–381. Springer, 2005.

[XN05] T. Xie and D. Notkin. Automatically identifying special and common unit tests for object-oriented programs. In *16th IEEE International Symposium on Software Reliability Engineering, 2005. ISSRE 2005*, page 11, 2005.

[XN06] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering*, 13(3):345–371, 2006.

[ZAD+10] Yuanyuan Zhang, Enrique Alba, Juan J. Durillo, Sigrid Eldh, and Mark Harman. Today/future importance analysis. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO '10)*, pages 1357–1364, Portland, USA, 7-11 July 2010. ACM.

[ZFH08] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search based requirements optimisation: Existing work & challenges. In *Proceedings of the 14th International Working Conference, Requirements Engineering: Foundation for Software Quality (RefsQ '08)*, volume 5025, pages 88–94, Montpellier, France, 16-17 June 2008. Springer.

[ZH10] Yuanyuan Zhang and Mark Harman. Search based optimization of requirements interaction management. In *Proceedings of the 2nd International Symposium on Search Based Software Engineering (SSBSE '10)*, pages 47–56, Benevento, Italy, 7-9 September 2010. IEEE.

[Zha10] Yuanyuan Zhang. *Multi-Objective Search-based Requirements Selection and Optimisation*. PhD thesis, King's College London, UK, 2010.

[ZHM07] Yuanyuan Zhang, Mark Harman, and S. Afshin Mansouri. The multi-objective next release problem. In *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO '07)*, pages 1129–1137 (Best Paper Award), London, UK, 7-11 July 2007. ACM.

[ZLT01] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical report, Swiss Federal Institute of Technology (ETH) Zurich, May 2001.