TestFul: an Evolutionary Test Approach for Java

Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz

Dipartimento di Elettronica e Informazione – Politecnico di Milano
piazza Leonardo da Vinci, 32
20133 - Milano (Italy)
email: {baresi,lanzi,miraz}@elet.polimi.it

***Abstract*—This paper presents *TestFul*, an evolutionary testing approach for Java classes that works both at class and method level. *TestFul* exploits a multi-objective evolutionary algorithm to identify the "best" tests. The paper introduces the main elements of *TestFul*. It also compares *TestFul* against well-known search-based solutions using a set of classes taken from literature, known software libraries, and independent testing benchmarks. The comparison considers statement and branch coverage, size of generated tests, and generation time. On considered classes, *TestFul* generates better tests than other search-based solutions, and achieves higher structural coverages with tests small enough to be usable.**

***Keywords*-Test Generation, Evolutionary Algorithms, Stateful Systems, Object-Oriented Paradigm, Multi-Objective Optimization**

I. Introduction

Search-based techniques do not concentrate on solving problems, but scan the space of possible solutions to identify the "best" ones. Some 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 [1]. Search-based test generation approaches have already been used to reveal failures in widespread systems [2]. 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 problems. 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., [3]) need several runs to create a complete test suite for the class under test. Moreover, targeting each element of the control flow graph separately could be misleading or provide insufficient guidance. For example, Ferguson [4] shows that a condition may depend on others, and this dependency may not be explicit in the control flow graph (but it requires a combined analysis of the control and data flow graphs to highlight it). If we target these conditions separately from their dependencies, we waste effort and reduce the overall efficiency of the approach.

To save on effort, we must also 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.

These considerations about efficiency, guidance, and reuse are the underpinnings of *TestFul*, our proposal for testing Java classes. *TestFul* leverages on search-based techniques and works at class level, to make the internal state on an object evolve, and at method level, to reach high coverage ratios.

The paper introduces the key elements of *TestFul* and presents a first empirical evaluation of its effectiveness. The benchmark comprises 15 classes taken from literature, public software libraries, and well-known benchmarks [5]. The comparison is against some promising search-based test generation approaches able to cope with stateful systems, namely *jAutoTest* (a Java-version of AutoTest [6]), *randoop* [7], and *etoc* [3]. Although limited to considered classes, the experiments highlighted that *TestFul* generates tests with higher statement and branch coverages than our competitors. Moreover, the small amount of time required to replay the generated tests eases their reuse during normal development, ensuring that newer versions correctly provide all the behaviors that must be preserved.

The rest of the paper is organized as follows. Section II introduces evolutionary algorithms to provide the reader with background concepts. Section III presents *TestFul*, while Section IV introduces related approaches, Section V describes the experiments, used in Section VI to empirically compare *TestFul* against some of the surveyed proposals. Section VII concludes the paper.

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

II. 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 [8]) may be reassembled to produce potentially better candidates. Mutation applies small random changes to the individuals in the 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.

Algorithm 1 Evolutionary Algorithm.

```
1: P ← RandomPopulation();
2: while Criterion Not Met do
3:     $\mathrm{P}_f$ ← EvaluateFitness(P);
4:     $\mathrm{P}_s$ ← Select($\mathrm{P}_f$);
5:     $\mathrm{P}_\chi$ ← ApplyRecombination($\mathrm{P}_s$, $p_\chi$);
6:     $\mathrm{P}_\mu$ ← ApplyMutation($\mathrm{P}_\chi$, $p_\mu$);
7:     P ← $\mathrm{P}_\mu$;
8: end while
```

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. 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.

III. TestFul

TestFul[1] is our framework for the automatic generation of unit tests for Java classes.

Stateful systems are particularly tedious when we want to test particular behaviors. 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 [9] 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. Conversely, it has been noticed that reaching a proper configuration of the object state can be expensive. Once the desired configuration is achieved, it can enable other behaviors, and new state configurations can be reached starting from the current one. A smart test-generation framework should both reuse states to exercise different behaviors, and recognize those new states that are useful to exercise new behaviors.

These are the underpinnings of *TestFul*, which extends previous approaches by considering also the internal state of objects to drive the exploration of the search space. Its structure (Figure 1) is inspired by the generic structure of tests for stateful systems, hence it is composed of two parts: one works on the state of objects, and the other focuses on exercising a particular function.

[1] It is released as open source software, and it is available at http://code.google.com/p/testful/.

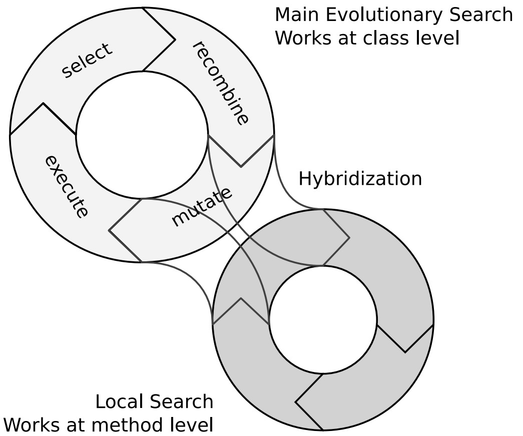


Figure 1. Two loops of *TestFul*.

The main evolutionary search module (depicted in light gray) works at class level, and 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 structural coverage (both statement and branch coverage) that the test achieves: the higher it is, the more likely the test is able to reach interesting states.

The local search module (dark gray) works at method level: it uses objects prepared by the main evolutionary search, and invokes a method of the class and exercises a particular behavior given the actual parameters and current state of the object. This operation is performed as a local search, which hybridizes the evolutionary algorithm used at class level. The local search targets an uncovered branch, that may represent a behavior not exercised yet. To drive the local search, we cannot use the structural coverage, since it would not provide any guidance. Instead, we focus on the condition that contains (i.e., controls directly) the targeted branch, and we monitor the values used. This information enables one to design a fitness function that provides enough guidance to the search process. If the local search is successful, it generates a test that exercises a branch of the method never executed previously: its structural coverage is higher, and it may be able to put objects in interesting states. Consequently the evolutionary algorithm at class level recognizes its importance, and uses it to reach even higher structural coverage.

A. Test Representation

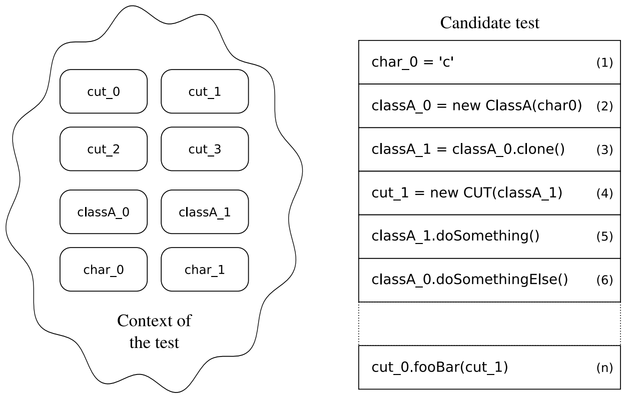
At the beginning, *TestFul* analyzes the class under test (CUT) to figure out all classes that might be involved in the test (the *test cluster*). This is done by considering the CUT and by transitively including the type of all parameters of all public methods (and constructors). Since abstract classes 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.

For each type contained in the test cluster, *TestFul* creates a set of variables, which we call *context* of the test. 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 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.

TestFul renders a test (i.e., an individual of both the evolutionary algorithm and the local search) as a sequence of operations that works on the context (see Figure 2). Each test starts from an initial context in which variables containing a reference are set to null, and those containing a primitive value are not initialized. Each operation of the test uses primitive values and objects —including instances of the class under test— taken from variables in the context, and also saves the results in these variables. This way, a test can make the values and objects in the context evolve, and reach complex 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



char_0 = 'c'	(1)
classA_0 = new ClassA(char0)	(2)
classA_1 = classA_0.clone()	(3)
cut_1 = new CUT(classA_1)	(4)
classA_1.doSomething()	(5)
classA_0.doSomethingElse()	(6)
cut_0.fooBar(cut_1)	(n)

Figure 2. Representation of Tests.

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.

B. Working at class level

When working at class level, we seek for a test able to put objects in interesting states. In order to recognize them, we use as heuristic the level of structural coverage that each test achieves. The higher it is, the more the test is able to exercise involved classes, thus the more likely it puts objects in interesting states. In particular, we adopt both branch and statement coverage. Note that the latter is able to detect behaviors not covered by the former, such as handling an exception.

These coverage values are only measurable by executing the test on an instrumented version of the class; however, this operation is extremely time-consuming, especially when a long test is being evaluated. To increase performance, we want the test to be *compact*: given two tests with the same ability to exercise the system, we favor the shortest one.

Summarizing, the fitness function *f(t)* of a test *t* consists of three parts:

$$f(t) = \langle t.length, stmtCov(t), brCov(t) \rangle$$

where *t.length* is the size of the test, measured as number of operations, and must be minimized; *stmtCov(t)* and *brCov(t)* represent respectively the statement and branch coverage, and must be maximized.

Recombination operators mix tests, reassembling sequences of operations able to reach interesting states. For this reason, during the recombination, we manage each operation as-is, without changing the objects it refers to or the values it uses.

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.

C. Working at method level

Working at class level does not provide enough guidance to cover all possible branches. For this reason, we hybridize the evolutionary algorithm by introducing a local search step that integrates the genetic operators. After some experiments, we decided to run the local search every 20 iterations of the evolutionary algorithm at the class level. We pick the best element of the current population, and focus on 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 and 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.

For each reachable but not exercised branch, we store the number of attempts done, and each local search focuses on those with fewer attempts. When the local search fails to reach a branch, we increase its number of attempts, and thus the next local search may prefer other branches. In several cases —for example, when a flag variable is used— covering other branches may also ease covering the current one. In other cases, reaching the targeted branch simply requires more effort than the given one. This is why we adjust the maximum number of iterations of the local search according to the number of attempts done.

Among branches with the same number of attempts, we pick the easiest to reach. For this purpose, we focus on the condition implied by each branch. If one of the elements being compared is a parameter of the method, it is possible to modify its value directly, thus the easiness of the branch increases. Moreover, some conditions provide more guidance than others, thus branches belonging to the former are easier to reach than those belonging to 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, we simplify the sequence

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
```

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 branch condition. If the local search is fruitful, we enrich the original test by appending the list of operations able to exercise the targeted branch. This way the structural coverage of the test increases, and the evolutionary algorithm working at class level will spread the new operations on the entire population.

As for the local search, we employ a simple hill climbing. This creates a mutated version of the test, and verifies if its execution is closer to execute the targeted branch 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 targeted branch is executed 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. 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 [10]. 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:

$$distance(a \oplus b) := \begin{cases} +\infty \leftrightarrow condition\ not\ executed \\ -\infty \leftrightarrow target\ executed \\ |a-b| \leftrightarrow otherwise \end{cases}$$

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. This way the search converges faster, and the desired branch is reached earlier.

IV. Related Work

The body of work on the automatic generation of functional tests is vast [11], and the limited space forces us to restrict the analysis by only considering search-based techniques, organized in two main groups: *blind* and *guided* search.

Blind search. Random testing [12], [13] 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 [14]. When failures are detected, random testing tools are able to provide witnesses, which are sequences of operations able to reveal the failure. 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 way, one obtains the same level of confidence on the system achieved through random testing. Unfortunately, this requires a huge amount of time, which precludes its applicability for regression testing.

Among available approaches, we mention AutoTest [15], 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 [16] to spot if an output value, or the state of an object, is incorrect, and thus reveals a failure [6]. Ciupa et al. [2] 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 [17] to generate

[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.

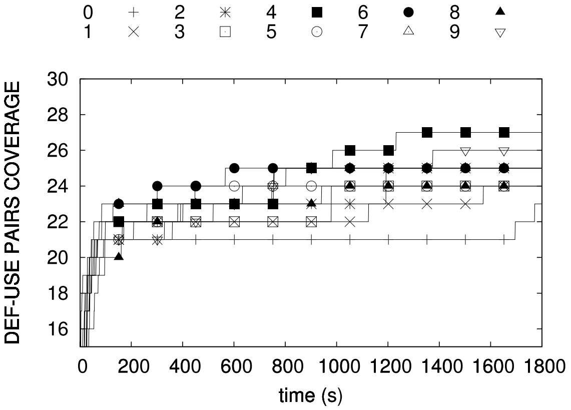


Figure 3. Def-Use pairs coverage with random testing. Complete coverage consists of 100 def-use pairs.

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 [18]. 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 [2]), measuring the structural coverage. Figure 3 shows the coverage of the actual 100 def-use pairs. Even with a powerful random number generator, random testing cannot converge to the complete coverage of the 100 def-use pairs. These results are coherent with those in [19], 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 [20]. 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 of failures.

Other approaches, such as [21], enhance traditional random test with taboo-search. They generate tests incrementally by adding to a previous test a randomly-chosen operation. They analyze each test, categorizing it as *error-revealing*, *new*, or *illegal*. Error-revealing tests are prompted to the user, and they are not used to create new sequences anymore. New tests are able to create objects not equivalent to those created in previous tests. They are output to the user for regression testing (they represent a normal behavior of the system), and are used as basis for generating new sequences. Illegal tests contain an illegal operation that violates the invoked method's preconditions. Since the system under test has no contracts, they employ a simple heuristic to recognize methods that do not accept null values as actual parameters. Those tests are discarded and are not used for generating new sequences.

Guided search. Other search-based test generation techniques are guided since the search process is directed towards the satisfaction of a goal. These techniques are extremely flexible, and were successfully applied to testing functional and non-functional properties. For example, Briand et al. [22] focus on real time systems; the goal is to find tests in which tasks miss their deadlines.

Focusing on structural testing, the goal is to reach the maximum coverage for a given criterion (e.g., cover all branches in the system). McMinn [9] detects two main approaches: coverage-oriented and structure-oriented. The former rewards the tests that cover more structural elements; for example, Watkins [23] 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. Instead, we focus on stateful systems, using the coverage level as heuristic for the number of object's states the test is able to achieve. Moreover, we hybridize the evolutionary algorithm with structure-oriented information, being able to provide guidance to execute the targeted branch.

Structure-oriented approaches tackle separately each uncovered structural element identified by the coverage criterion [24]. 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 each structural element separately from the others 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 of the test to reach the desired structural element [1], [10]. 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 leads to the target. These proposals are able to guide the search process successfully towards the selected target, but they are not able to deal with dependencies not reported in the control flow graph. As an example, they must be extended to cope with flag variables: in this case one must also analyze the data flow graph of the program as proposed in the chaining approach [4] and subsequent refinements [25]. 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.

Moreover, these proposals work on stateless systems: they consider a single function invocation and generate the input parameters to reach the selected structural element. In contrast, Tonella [3] focuses on object-oriented systems, and 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, its 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 starts from scratch.

V. Design of Experiments

We compared *TestFul* against some promising search-based approaches able to work on stateful systems, namely *jAutoTest* (a version of AutoTest [6], developed for Java), *randoop* [7], and *etoc* [3]. *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 able to 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, listed in Table I. 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 [28] 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* [29], 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, and calculated its average performance by excluding those two classes. Moreover, the memory required by *randoop* increased with the duration of 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 generated tests.

To evaluate the approaches instead of the tools, we discarded the results when we got errors (e.g., out of memory[3]

Name & Provenience	**LOC**	**Cyclomatic complexity**
Array Partition [5]	49	3.40
Binary Heap [5]	79	3.27
Binary Search Tree [5]	156	2.42
Coffee Vending Machine [26]	31	1.29
Doubly Linked List [5]	356	2.60
Disjoint Set [5]	53	4.00
Disjoint Set Fast [5]	59	4.60
Fraction [27]	319	3.91
Hard State Machine	48	10.00
Red Black Tree [5]	507	3.70
Sorting [5]	179	2.81
Stack Array [5]	53	1.80
Stack List [5]	76	1.92
Simple State Machine [28]	27	10.00
Vector [5]	326	2.33

Table I
Benchmark.

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

or non-termination[4]) 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 materializes only on classes taken from the independent repository, and it is due to two reasons. First, they contain some unreachable code, such as some private utility methods never used in the class. Second, those classes declare protected or friendly methods never invoked explicitly. Note that all approaches acted 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.

VI. Experimental Results

This section only provides a summary of the experiments made[5]. For each time limit we used and for each of the four tools, Table II 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.

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 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, which decreases with long runs (this happens only with *TestFul*). This phenomenon 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.

(a) 5-minute runs.

Tool	Size of Tests	Statement Coverage	Branch Coverage
	$\mu(s)$ LOC	$\mu(s)$ %	$\mu(s)$ %
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	Statement Coverage	Branch Coverage
	$\mu(s)$ LOC	$\mu(s)$ %	$\mu(s)$ %
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	Statement Coverage	Branch Coverage
	$\mu(s)$ LOC	$\mu(s)$ %	$\mu(s)$ %
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	Statement Coverage	Branch Coverage
	$\mu(s)$ LOC	$\mu(s)$ %	$\mu(s)$ %
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)*

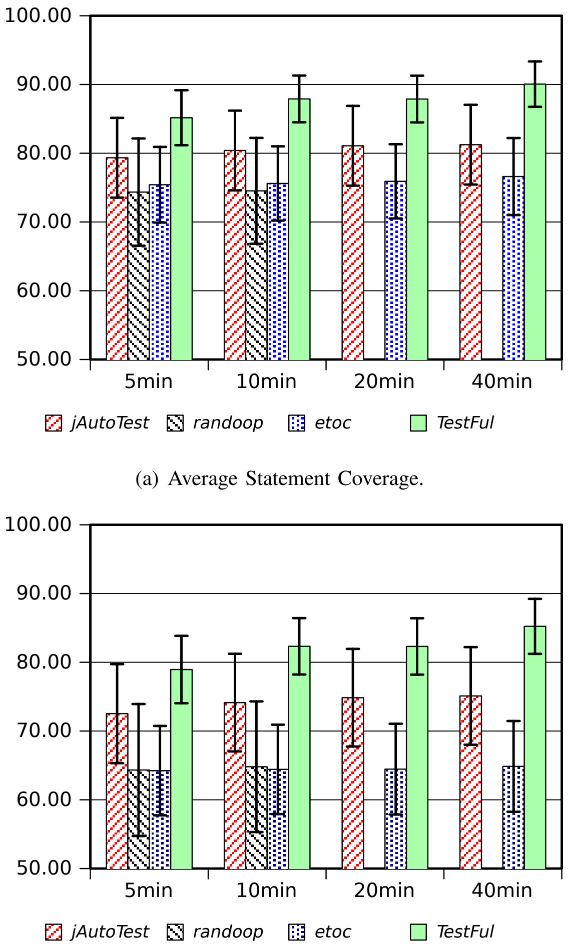
Table II
Performances.

As for the size of generated tests, *TestFul* creates a test suite smaller than *randoop* and *jAutoTest*, but bigger than *etoc*. However, the tests it generates are usable 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, hardly usable as-is for regression testing (they would require several minutes to run).

It is also interesting to analyze the possible relationship between the size of tests and the length of the runs for the four tools. *Randoop* shows a direct relationship: the size of 10-minute runs are slightly more than the double of the size of 5-minute runs. This phenomenon forbids 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 must consider the problem of *bloat* [30]. 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 tries to minimize the length of individuals, but its overall contribution is limited. We might increase its weight, but this way we would

[4] 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*.

[5] 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. Average structural coverage.

decrease the exploration of the solution space, pushing the search towards shorter tests, and lowering the capability of *TestFul* to generate good tests. Since the size of the result was acceptable, we decided to leave the fitness function as-is; the user may always adopt some analysis techniques (e.g., def-use analysis, slicing, or delta debugging) to produce a smaller test, as shown in [31].

Figure 5 compares branch coverage against time limits to understand whether the approaches achieve better results with longer runs (statement coverage has a similar trend). The more available time increases, the more thoroughly *TestFul* exercises the class under test. Similarly, *jAutoTest* shows a less-evident increasing trend. Conversely, other tools' performances remain almost stable. This phenomenon can be explained by considering that *jAutoTest* and *TestFul* have a similar internal test representation. It fosters the evolution of objects' states, thus longer runs reach more complex configurations and achieve higher structural coverages.

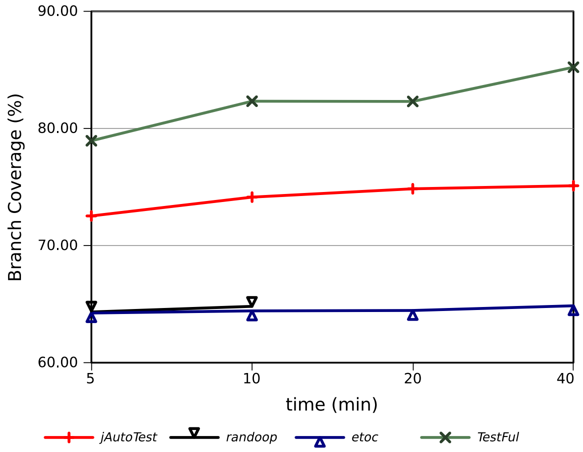


Figure 5. Branch Coverage vs. time limit.

VII. Conclusions and Future Work

This paper introduces *TestFul*, a novel framework for the automatic unit test generation for Java classes based on a hybrid multi-objective evolutionary optimization. Compared to the state of the art in search-based test generation, *TestFul* recognizes and reuses useful state configurations to exercise different features. This is combined with a guidance able to reach uncovered structural elements and leads to overall good performances. Moreover, it uses a multi-objective evolutionary algorithm to evolve candidate tests by combining the structural coverage with the compactness of the tests. This ensures high efficiency.

We compared *TestFul* against well-known search-based approaches. As benchmark, we selected a set of classes taken from literature, real-world applications, and other independent benchmarks. Obtained results, although limited to the considered problems, highlight the validity of the approach, 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 a complete lack of guidance (*jAutoTest* and *randoop*) or a partial one (*etoc*). 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.

Our future work comprises the refinement of the search strategy. We plan to consider other analysis techniques, such as the data flow, and also to understand how to exploit the domain knowledge to improve the search strategy by identifying more precisely the interesting states of the different objects. In parallel, we would like to use *TestFul* with other stateful systems.

References

[1] C. C. Michael, G. McGraw, and M. Schatz, “Generating software test data by evolution,” *IEEE Trans. Software Eng.*, vol. 27, no. 12, pp. 1085–1110, 2001.

[2] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, “Experimental assessment of random testing for object-oriented software,” in *ISSTA*, D. S. Rosenblum and S. G. Elbaum, Eds. ACM, 2007, pp. 84–94.

[3] P. Tonella, “Evolutionary testing of classes,” in *ISSTA*, G. S. Avrunin and G. Rothermel, Eds. ACM, 2004, pp. 119–128.

[4] R. Ferguson and B. Korel, “The chaining approach for software test data generation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 1, pp. 63–86, 1996.

[5] H. Do, S. G. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.” *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[6] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva, “Contract driven development = test driven development - writing test cases,” in *ESEC/SIGSOFT FSE*, I. Crnkovic and A. Bertolino, Eds. ACM, 2007, pp. 425–434.

[7] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *29th International Conference on Software Engineering (ICSE 2007)*, 2007, pp. 75–84.

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

[9] P. McMinn, “Search-based software test data generation: a survey,” *Softw. Test., Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

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

[11] M. Pezzè and M. Young, *Software testing and analysis*. Wiley, 2008.

[12] D. Hamlet, “When only random testing will do,” in *Random Testing*, J. Mayer and R. G. Merkel, Eds. ACM, 2006, pp. 1–9.

[13] R. Hamlet, “Random testing,” in *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[14] T. Y. Chen and Y.-T. Yu, “On the expected number of failures detected by subdomain testing and random testing,” *IEEE Trans. Software Eng.*, vol. 22, no. 2, pp. 109–119, 1996.

[15] B. Meyer, I. Ciupa, A. Leitner, and L. Liu, “Automatic testing of object-oriented software,” in *Proceedings of SOFSEM 2007: 33rd Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 2007, pp. 114–129.

[16] B. Meyer, “Applying ‘design by contract’,” *Computer*, vol. 25, no. 10, pp. 40–51, Oct 1992.

[17] D. E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1997, vol. 2: Seminumerical Algorithms, ch. 3.2.1: The Linear Congruential Method, pp. 10–26.

[18] M. Matsumoto and T. Nishimura, “Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[19] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer, “On the predictability of random tests for object-oriented software,” in *International Conference on Software Testing, Verification, and Validation (ICST)*. IEEE Computer Society, 2008, pp. 72–81.

[20] T. Chen, H. Leung, and I. Mak, “Adaptive Random Testing,” in *Advances in Computer Science - ASIAN 2004*, Springer, Ed., vol. 3321/2005, 2004, pp. 320–329.

[21] C. Pacheco, S. K. Lahiri, and T. Ball, “Finding errors in .net with feedback-directed random testing,” in *ISSTA*, B. G. Ryder and A. Zeller, Eds. ACM, 2008, pp. 87–96.

[22] L. C. Briand, Y. Labiche, and M. Shousha, “Stress testing real-time systems with genetic algorithms,” in *GECCO*, H.-G. Beyer and U.-M. O’Reilly, Eds. ACM, 2005, pp. 1021–1028.

[23] A. Watkins, “The automatic generation of test data using genetic algorithms,” in *In Proceedings of the Fourth Software Quality Converence*, 1995, pp. 300–309.

[24] B. Korel, “Automated software test data generation,” *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[25] P. McMinn and M. Holcombe, “Evolutionary testing using an extended chaining approach,” *Evolutionary Computation*, vol. 14, no. 1, pp. 41–64, 2006.

[26] U. A. Buy, A. Orso, and M. Pezzè, “Automated testing of classes,” in *ISSTA*, 2000, pp. 39–48.

[27] Apache Software Foundation, “The apache commons mathematics library,” http://commons.apache.org/math/.

[28] R. Majumdar and K. Sen, “Hybrid concolic testing,” in *29th International Conference on Software Engineering (ICSE 2007)*, 2007, pp. 416–426.

[29] “Cobertura,” http://cobertura.sourceforge.net/.

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

[31] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, “Efficient unit test case minimization,” in *ASE*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, 2007, pp. 417–420.