Improving Evolutionary Testing by Means of Efficiency Enhancement Techniques

Matteo Miraz, Pier Luca Lanzi, *Member, IEEE* and Luciano Baresi

***Abstract*— *TestFul* is a novel evolutionary testing approach for object-oriented programs with complex internal states. In our preliminary experiments, it already outperformed some of the well-known search-based testing approaches. In this paper we show how *TestFul* can be further improved by leveraging three efficiency enhancement techniques: seeding, hybridization, and fitness inheritance. We considered four extensions of *TestFul*: three using each enhancement separately, and one using all of them at the same time. We used these new versions of *TestFul* to generate tests for six Java classes taken from the literature, public software libraries, and third party benchmarks. We compared the performance of the original *TestFul* against these new versions. Our results show that each enhancement technique results in a significant speed-up and, even more interesting, the highest improvement is achieved when all the enhancements are combined together.**

I. Introduction

Thorough testing should be a fundamental activity of any software development process. Even if a careful testing process cannot guarantee the delivery of error-free systems, it would definitively help improve the quality of the final product and increase the confidence on its correctness. Several studies confirm the importance of this activity: for example, Tassey [1] estimated that $20 billions could be saved every year if better testing were performed.

Tests that exercise single classes, isolated from the rest of the system, are called *unit tests*, which are widely used given their ability to detect errors. 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, tests must conform to some adequacy criteria. 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 tests. For example, the *branch coverage* is 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 more deeply the test exercises the system, hence the more likely it is able to discover the errors it may contain.

However, testing a software system is an expensive activity, which can accounts for 50% of the overall cost of the development cycle [2]. Several approaches have been proposed to reduce these costs through automatic test generation. In particular, *evolutionary testing* [3] recently achieved significant results and appears to be one of the most promising approaches for the automatic generation of tests [4].

Evolutionary testing models the test-generation process as a search problem: individuals represent tests (i.e., sequences of statements); evolutionary computation is applied to search for the best tests (i.e., the sequences of statements which can highlight the highest number of errors). The vast majority of evolutionary testing methods apply a *divide and conquer* paradigm: once a coverage criterion is chosen (e.g., branch coverage), they identify a set of sub-goals that must be satisfies (e.g., each branch in the control flow graph), and target each of them separately. In [5], [6], we already showed that divide and conquer approaches may result in a dramatic waste of computational effort when applied to object-oriented systems with complex internal states. This is why we introduced *TestFul*, a *holistic* approach that (i) does not require the identification of sub-goals and therefore (ii) it works on the entire test-generation task (and not just on separate elements). We also applied *TestFul* to generate tests for Java classes (taken from the literature, public software libraries, and third party benchmarks) and compared its performance against other well-known search-based approaches. The first results show that *TestFul* can generate tests able to reach higher statement and branch coverages than other divide and conquer approaches.

In this work, we want to move *TestFul* a step further and improve its evolutionary engine by enriching it with advanced efficiency enhancement techniques (EETs). In particular, we consider three types of efficiency enhancements: *seeding*, *hybridization*, and *fitness inheritance*. Seeding improves the quality of the initial population to speed up the early stage of the evolutionary process (by providing better building blocks right from the start). Hybridization enriches typical (blind) mutation with local search that, by levering domain-specific information, can significantly improve the individuals (i.e., the candidate test). Fitness inheritance replaces, for some of the individuals in the population, an expensive fitness evaluation with an approximated fitness that is computed (inherited) from similar individuals.

We performed a set of experiments to test the actual improvement of the three efficiency enhancements on a benchmark of six Java classes, taken from the literature, public software libraries, and third party benchmarks. We tested each enhancement separately and also altogether. Results show that seeding, hybridization, and fitness inheritance significantly improve the performance of *TestFul* both when

Luciano Baresi, Pier Luca Lanzi and Matteo Miraz are with the Politecnico di Milano, Dipartimento di Elettronica e Informazione, Piazza Leonardo da Vinci 32, I-20133 Milano, Italy; Email: {baresi,lanzi,miraz}@elet.polimi.it. This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

they are applied separately and altogether. The highest improvement is obtained when all the enhancements are applied at the same time.

The paper is organized as follows. Section II introduces *TestFul* while Section III introduces the methods of efficiency enhancement we used. Section IV describes the experimental set up. Section V presents the results of the evaluation we carried out on each enhancement separately and for the three methods altogether. Section VI surveys related approaches and Section VII concludes the paper.

II. TestFul

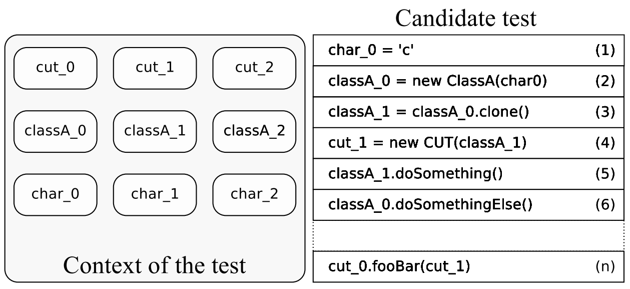
Stateful software systems, that is, systems with internal state, are extremely difficult to test since they require that objects be put in particular states before their functionality can be tested. One must first put objects in proper states and then provide the correct values for the input parameters of the behaviors to be tested. Arcuri [7] noted that reaching a proper state of the object under test can be expensive. In addition, once the desired configuration is achieved, it usually enables other behaviors, and new state configurations can be reached starting from the current one. This means that 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.

TestFul[1] is our framework for the automatic generation of unit tests for Java classes. It extends previous approaches by taking into account the internal state of objects to drive the exploration of the search space.

Test Representation. *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, called *context of the test*. To enable polymorphism, *TestFul* 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 the interfaces implemented by `A`). 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 work on the context (see Figure 1). 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 the 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 make them reach complex configurations.

[1]*TestFul* is available at `http://code.google.com/p/testful/`.



Candidate test
cut_0
cut_1
cut_2
classA_0
classA_1
classA_2
char_0
char_1
char_2
Context of the test
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)

Fig. 1. Test representation.

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 stores its reference in a variable of the context. As for parameters, it uses objects and primitive values taken from the variables in the context. If one of the parameters has a primitive type and the variable is not initialized, the operation is not valid and it is skipped.
- **invoke** invokes a method. The receiving object and actual parameters are taken from the 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 is not initialized, the operation is skipped since it is not valid.

Evolutionary Testing. *TestFul* employs an evolutionary algorithm to search for the tests able to reach all the interesting internal states. To recognize them, we use the level of structural coverage (both statement and branch coverage) that each test achieves. The higher it is, the better a test is able to exercise a class, and thus it should put objects in interesting states.

These coverage values are only measurable by executing the test on an instrumented version of the class, but 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.

To summarize, 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, statement and branch coverages, and they must be maximized.

Recombination operators mix tests, reassembling sequences of operations able to reach interesting states. For this reason, during 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 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.

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.

III. Efficiency Enhancement Techniques

Our previous work [5] shows that *TestFul* can generate short tests for Java classes with complex internal states. Those experiments used classes taken from the literature, public software libraries, and third party benchmarks, and showed encouraging results. By leveraging powerful multi-objective evolutionary search, *TestFul* was able to generate better tests than other search-based approaches.

However, we also noted [5] that search-based and evolutionary testing are still computationally expensive and require a significant amount of CPU time. This is why this paper focuses on improving the evolutionary engine of *TestFul* through the use of modern efficiency enhancement techniques (EETs) [8], [9]. In particular, we focused on three classes of techniques: *seeding*, *hybridization*, and *fitness inheritance*.

Seeding. In [5], [6], *TestFul* considered an initial population of randomly generated tests. However, later analyses revealed that 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 [9].

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 [5], we showed that random test usually finds good tests in the very first moments of its application. 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.

Hybridization. *TestFul* recombines tests that put objects in different states to generate tests that exercise new features. Some classes however have features only exercisable via precise sequences of invocations. If the population does not contain all the invocations needed, recombination and mutation can hardly generate a suitable test, quickly and reliably.

To mitigate this issue, and speed up the search, we hybridized the evolutionary algorithm of *TestFul* by introducing a mutation operator based on a step of local search. Note that, we run local search only on a small portion of individuals. In particular, we considered three policies: *best*, *5%*, and *10%*. The first one only selects the best test found so far; the other ones randomly select the 5% or 10% of individuals from the entire population.

The local search targets a reachable but uncovered branch, which may represent a functionality 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. We only focus on these types of branches because they are easy to reach (we only have to change the outcome of a single condition) and they require less effort than the more advanced methods used in [3]. In fact, other branches would require much more effort, more conditions should be considered, and would also result in a huge overhead for the system.

The local search focuses on the selected branch, and uses a simple hill climbing to modify the test and execute the selected branch. To modify the test, it randomly picks an operation and: (i) *removes* it, or (ii) *adds* a random operation before or after it, or (iii) *changes 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 might also try to change the stored value. We add a random value to the original one, or flip its value if it is a boolean variable. To drive the local search process, we use simplified version of the fitness used in [10] as scoring metric.

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., [8], [11]).

Name & Provenience	LOC	Cyclomatic complexity
Disjoint Set Fast [12]	59	4.60
Fraction [13]	319	3.91
Red Black Tree [12]	507	3.70
Sorting [12]	179	2.81
Stack Array [12]	53	1.80
State Machine [14]	27	10.00

TABLE I
BENCHMARK.

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.

IV. DESIGN OF EXPERIMENTS

To evaluate the improvement in performance introduced by the use of efficiency enhancements, we devised a benchmark consisting of the six classes of Table I, and compared the orignal version of *TestFul* [5], [6] 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 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.

Disjoint Set (Fast). This class handles dynamic partitions of a set *X* composed of the first *n* integers, where *n* is specified when the object is instantiated. The class 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 [12] provide two implementations of this class: we choose the one marked as *fast*, which is more challenging.

Fraction. This is an immutable class that represents fractions and provides the basic operations between them. This class is taken from the Apache Commons Maths library [13] and it is widely used in several products.

Red-Black Tree. The internal data structure of this class 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 the Sun's implementation of the class `java.utils.TreeMap`, which is used in several related work as benchmark.

Sorting. This class manages 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 Array. This class implements the stack data structure with LIFO (Last-In, First-Out) access policy. The implementation internally uses an array with a fixed capacity, checking 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).

The State Machine. This class is inspired by a function present in the VIM text editor [14]. 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.

V. EMPIRICAL RESULTS

We compared each enhancement against the original version of *TestFul* [5], [6] by using the benchmark of Table I. All the experiments discussed here were run on an Intel Core2 Quad CPU Q6600@2.40GHz with 4 gigabytes of RAM.

To evaluate the improvements, we first computed the average performance of the best individual during the evolution using the original and an enhanced version of *TestFul*. Figure 2 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 (Table II).

Seeding. Seeding leads to significant improvements in all the classes (ranging from 11.75% to a stunning 98.27%), but the *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 2(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 II). In [6], 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

Class	Seeding	Hybridization			Fitness inheritance		overall
		best	5%	10%	frontier	uniform	
Disjoint Set Fast	+53.79%	+43.94%	+35.61%	+28.13%	+36.45%	+5.84%	+54.85%
Fraction	+11.75%	-1.05%	-16.69%	-18.97%	-15.72%	-3.98%	+10.96%
Red-Black Tree	+98.27%	+64.78%	+24.01%	+6.07%	-18.57%	+26.88%	+90.12%
Sorting	+13.35%	+15.90%	+12.23%	+9.69%	+13.58%	+16.50%	+18.28%
Stack Array	+19.06%	+18.21%	+18.36%	+18.33%	+0.55%	+1.73%	+19.06%
State Machine	*same*	+399.97%	+347.49%	+315.39%	*same*	*same*	+472.60%
average	+32.70%	+90.29%	+70.17%	+59.77%	+2.72%	+7.83%	+110.98%

TABLE II
IMPROVEMENT WITH EACH EVOLUTIONARY ENHANCEMENT TECHNIQUE.

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 2(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 II), where we have already shown [6] 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*.

Hybridization. The data in Table II 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 2(c)).

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 2(d) for class *Disjoint Set Fast*.

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

Fitness Inheritance. 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 2(e) and 2(f) 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 on the frontier. The former might overestimate 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 likely generates tests that will 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 2(g)). 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 2(h)) 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.

Overall 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 II (column **overall**) 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 2(i)). However, they also allow *TestFul* to generate tests with higher quality (see the results for class *Sorting* in Figure 2(j)).

Note that different efficiency enhancement techniques can cooperate to increase the performance even more. This is the case of the *State Machine*: *hybridization* only improves performance of 400%, while *fitness inheritance* has no effect. However, when applied together, *hybridization* enables *fitness inheritance* to contribute more in increasing performance, reaching a speed-up of 473%.

VI. Related Work

To the best of our knowledge, nobody applied efficiency enhancement techniques to evolutionary testing. Probably, this is due to the *divide and conquer* approach generally used to tackle test generation. By focusing on reaching each structural element separately, the generation of a good initial population and the creation of a hybridization mechanism are hindered, and the approach is more susceptible to noise in the fitness function.

In contrast, *TestFul* uses a holistic approach to test generation, which also allows us to introduce efficiency enhancement techniques. For this purpose, we exploited principles behind other search-based approaches. This section briefly overviews these works, which are categorized in two main groups: *blind* and *guided* search techniques.

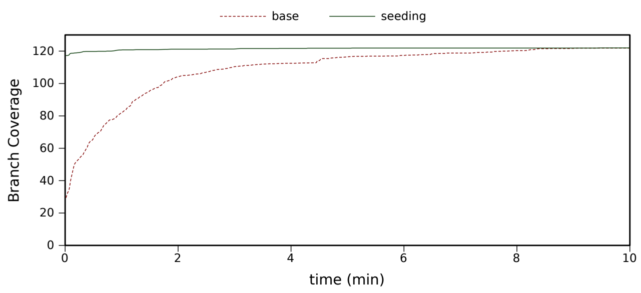
Blind search. Random testing [15] is probably the most famous search-based approach for the automatic generation of tests. It simply performs a random sequence of invocations on the system under test. Notwithstanding its simplicity, random testing can be as effective as other traditional approaches [16]. 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 the replay of the whole sequence of random operations. This operation requires a huge amount of time, but it allows one to obtain the same level of confidence on the system achieved through random testing.

Among available approaches, we mention *AutoTest* [17], one of the most advanced tools for random testing, specifically designed for object-oriented systems. By exploiting its principles, we created *jAutoTest*, a Java implementation that mainly differs from the original tool in the ability to generate a synopsis of executed tests. 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, and it is used by the *seeding* technique to create a better initial population.

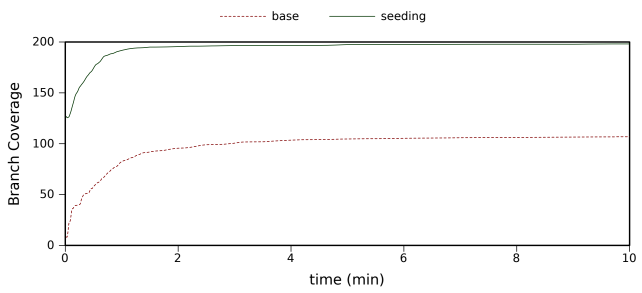
To augment effectiveness, some works also propose *adaptive random testing* (ART) to ensure that generated values are equally distributed over the input domain [18]. The idea is that the more distant values are, the better they are able to reveal failures. However, empirical studies show that ART tools do not discover failures earlier; instead they reveal a different set of failures. Other approaches, such as [19], enhance traditional random test with taboo-search. They generate tests incrementally by adding a randomly-chosen operation to a previous test. If a test is able to create objects not equivalent to those created in previous tests, it is used as basis 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. Usually, the goal is to reach the maximum coverage for a given criterion (e.g., cover all branches in the system). The approaches proposed so far tackle separately each uncovered structural element identified by the coverage criterion [20]. 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.

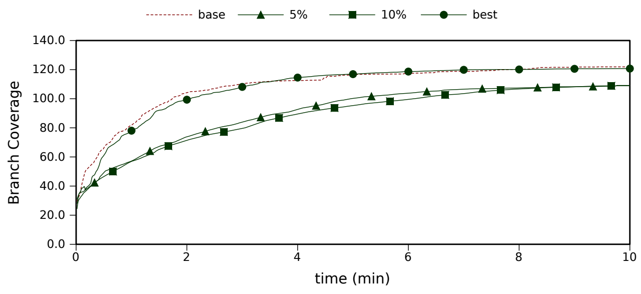
There are several works that refine the structure-oriented approach by proposing functions able to better guide the search process. Initially, they use the control flow graph of the program to judge the distance of the test to reach the desired structural element [10], [21]. By comparing the execution 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



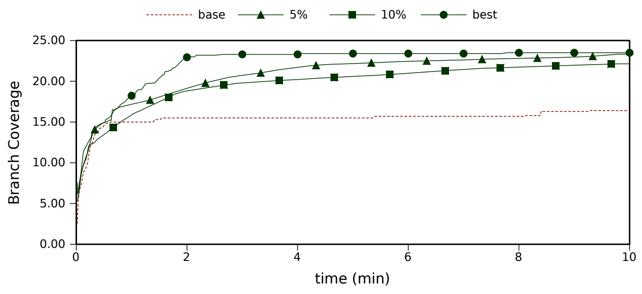
(a) Seeding: performance on class Fraction



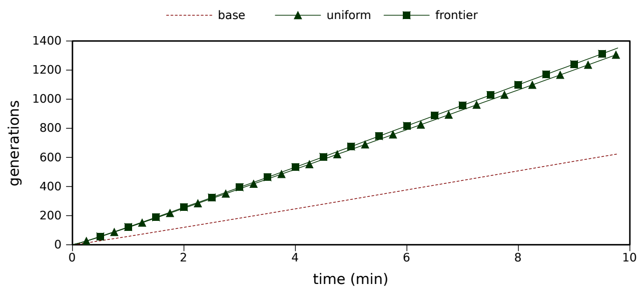
(b) Seeding: performance on class Red-Black Tree



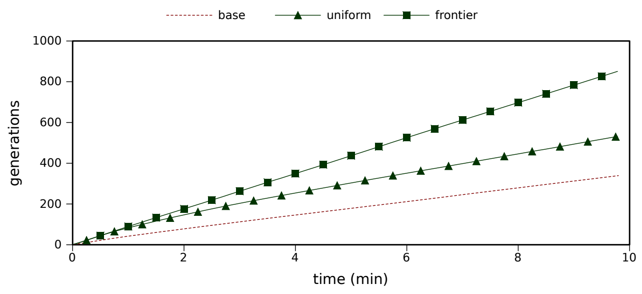
(c) Hybridization: performance on class Fraction



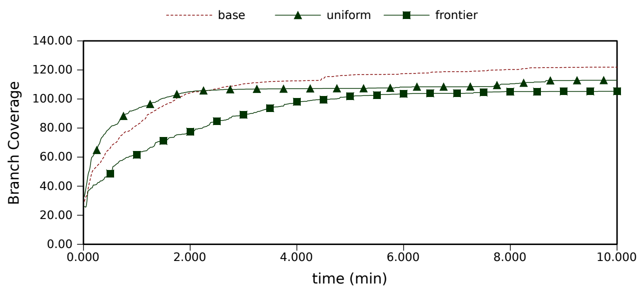
(d) Hybridization: performance on class Class Disjoint Set Fast



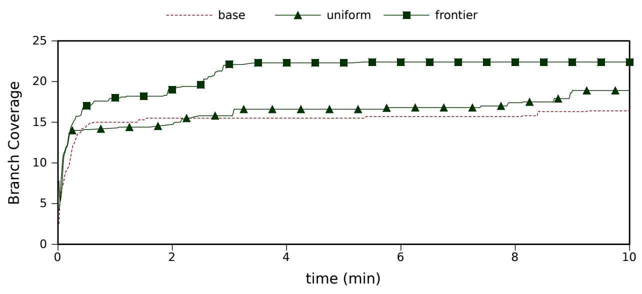
(e) Fitness Inheritance: speed-up on class Stack Array



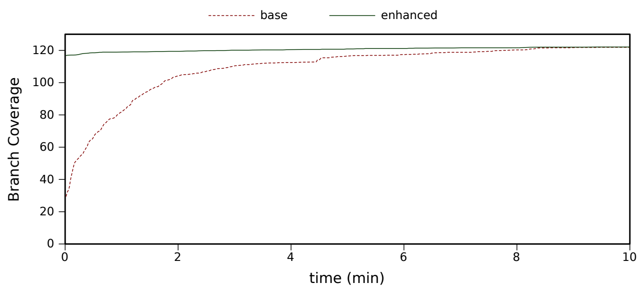
(f) Fitness Inheritance: speed-up on class Red-Black Tree



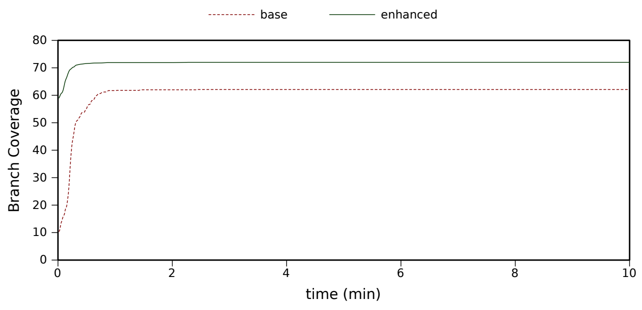
(g) Fitness Inheritance: performance on class Fraction



(h) Fitness Inheritance: performance on class Disjoint Set Fast



(i) Overall performance improvement on class Fraction



(j) Overall performance improvement on class Sorting

Fig. 2. Efficiency Enhancement Techniques.

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 [22] and subsequent refinements [23]. 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 the 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 re-starts from scratch.

The hybridization technique proposed in this paper is inspired by these works. It focuses on a single branch of the system, and modifies a test to execute it. However, we chose as starting element a test able to reach the condition that implies that branch, hence we can rely on a cheaper search algorithm.

VII. Conclusions and Future Work

This paper proposes an improved version of the evolutionary component of *TestFul*, our framework for the automatic generation of tests for Java classes. We considered three efficiency enhancement techniques: *seeding*, *hybridization*, and *fitness inheritance*. The first provides *TestFul* with a high-quality initial population that can speed up the initial part of the evolutionary process. The second hybridizes the evolutionary algorithm with a local search that can improve the discovery of new structural elements. The third speeds up the search process by replacing the evaluation of the fitness function of some individuals with an estimated fitness inherited from their parents.

We evaluated the different efficiency techniques on a set of classes taken from open-source libraries and other independent benchmarks. Our results, although limited to the considered problems, show a huge improvement of the test generation process. *TestFul* was able to achieve better results by using less computational resources.

Our future work includes a deeper exploration of the *seeding* technique to increase the performance of *TestFul* when applied on newer versions of a tested class. In this case, we plan to adapt the previously generated tests for the class to work with its newer version. Moreover, we also plan to investigate the possibility to seed the initial population for testing a class `C` with tests generated for classes similar to `C` (e.g., those implementing the same interfaces).

References

[1] G. Tassey, “The Economic Impacts of Inadequate Infrastructure for Software Testing,” National Institute of Standards and Technology RTI Project, Tech. Rep., 2002.

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

[3] P. Tonella, “Evolutionary Testing of Classes,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[4] P. McMinn, “Search-based Software Test Data Generation: a Survey,” *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[5] M. Miraz, P. L. Lanzi, and L. Baresi, “TestFul: Using a Hybrid Evolutionary Algorithm for Testing Stateful Systems,” in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2009, pp. 1947–1948.

[6] L. Baresi, P. L. Lanzi, and M. Miraz, “TestFul: an Evolutionary Test Approach for Java,” in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[7] A. 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.

[8] K. Sastry, “Evaluation-Relaxation Schemes for Genetic and Evolutionary Algorithms,” University of Illinois at Urbana-Champaign, Urbana, IL, Tech. Rep. 2002004, Feb. 2002.

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

[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] K. Sastry, “Genetic algorithms and genetic programming for multiscale modeling: Applications in materials science and chemistry and advances in scalability,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, 2007.

[12] 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.

[13] Apache Software Foundation, http://commons.apache.org/math/.

[14] “Vim,” http://www.vim.org.

[15] D. Hamlet, “When Only Random Testing Will Do,” in *Proceedings of Random Testing*, 2006, pp. 1–9.

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

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

[18] T. Chen, H. Leung, and I. Mak, “Adaptive Random Testing,” in *Proceedings of Advances in Computer Science — ASIAN*, Springer, Ed., vol. 3321/2005, 2004.

[19] C. Pacheco, S. K. Lahiri, and T. Ball, “Finding Errors in .NET with Feedback-Directed Random Testing,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2008, pp. 87–96.

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

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

[22] R. Ferguson and B. Korel, “The chaining approach for software test data generation,” *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 1, pp. 63–86, 1996.

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