Adaptive Evolutionary Testing: an Adaptive Approach to Search-Based Test Case Generation for Object-Oriented Software

José Carlos Bregieiro Ribeiro, Mário Alberto Zenha-Rela,
and Francisco Fernández de Vega

Abstract Adaptive Evolutionary Algorithms are distinguished by their dynamic manipulation of selected parameters during the course of evolving a problem solution; they have an advantage over their static counterparts in that they are more reactive to the unanticipated particulars of the problem. This paper proposes an adaptive strategy for enhancing Genetic Programming-based approaches to automatic test case generation. The main contribution of this study is that of proposing an Adaptive Evolutionary Testing methodology for promoting the introduction of relevant instructions into the generated test cases by means of mutation; the instructions from which the algorithm can choose are ranked, with their rankings being updated every generation in accordance to the feedback obtained from the individuals evaluated in the preceding generation. The experimental studies developed show that the adaptive strategy proposed improves the test case generation algorithm's efficiency considerably, while introducing a negligible computational overhead.

1 Introduction

The application of Evolutionary Algorithms (EAs) to test data generation is often referred to as Evolutionary Testing (ET) [Tonella(2004)] or Search-Based Testing (SBT) [McMinn(2004)]. ET is an emerging methodology for automatically generating high quality test data; it is, however, a difficult subject – especially if the aim is to implement an automated solution, viable with a reasonable amount of computa-

José Carlos Bregieiro Ribeiro
Polytechnic Institute of Leiria – Leiria, Portugal, e-mail: jose.ribeiro@estg.ipleiria.pt

Mário Alberto Zenha-Rela
University of Coimbra – Coimbra, Portugal, e-mail: mzrela@dei.uc.pt

Francisco Fernández de Vega
University of Extremadura, – Mérida, Spain, e-mail: fcofdez@unex.es

tional effort, which is adaptable to a wide range of test objects. This paper proposes an adaptive methodology for the ET of Object-Oriented (OO) software.

EAs are powerful, yet general, methods for search and optimization. Their generality comes from the unbiased nature of the standard operators used, which perform well for problems where little or no domain knowledge is available [Angeline(1995)]. However, if knowledge about a problem is available, a bias can be introduced directly into the problem so as to remove (or penalize) undesirable candidate solutions and improve the efficiency of the search. Unfortunately, *a priori* knowledge about the intricacies of the problem is frequently unavailable. Having little information about a problem does not, however, necessarily prevent introducing an appropriate specific bias into an evolutionary problem; for many tasks, it is possible to dynamically adapt aspects to anticipate the regularities of the environment and improve solution optimization or acquisition speed. Adaptive EAs are distinguished by their dynamic manipulation of selected parameters or operators during the course of evolving a problem solution [Hinterding et al(1997)]. They have an advantage over their standard counterparts in that they are more reactive to the unanticipated particulars of the problem and, in some formulations, can dynamically acquire information about regularities in the problem and exploit them.

Typically, EAs maintain a population of candidate solutions rather than just one current solution; in consequence, the search is afforded many starting points, and the chance to sample more of the search space than local searches. Mutation is the main process through which new genetic material is introduced during an evolutionary run with the intent of diversifying the search and escaping local maxima. The main contribution of this study is that of proposing an adaptive strategy for promoting the introduction of relevant instructions into the existing test cases by means of mutation; the set of instructions from which the algorithm can choose is ranked, with their rankings being updated every generation in accordance to the feedback obtained from the individuals evaluated in the preceding generation.

This article is organized as follows. The next Section starts by providing background on ET and Adaptive EAs. Section 3 details the Adaptive ET strategy proposed. The experiments conducted in order to validate and observe the impact of the Adaptive ET technique are discussed in Section 4. The concluding Section presents some final considerations and summarizes the most relevant contributions.

2 Background and Terminology

Software Testing is the process of exercising an application to detect errors and to verify that it satisfies the specified requirements. When performing unit-testing, the goal is to warrant the robustness of the smallest units – the *test objects* – by testing them in an isolated environment. Unit-testing is performed by executing the test objects in different scenarios using relevant *test cases*. Classes and objects are typically considered to be the smallest units that can be tested in isolation in OO programs [Wappler and Wegener(2006a)]. An object stores its state in fields and

exposes its behaviour through methods. A unit-test case for OO software consists of a *Method Call Sequence (MCS)*, which defines the test scenario. During test case execution, all participating objects are created and put into particular states through a series of method calls. Each test case focuses on the execution of one particular public method – the *Method Under Test (MUT)*. In most situations, an OO class is not standalone; testing a single class involves other classes – i.e., classes that appear as parameter types in the method signatures of the *Class Under Test (CUT)*. The set of classes which are relevant for testing a particular class is called the *Test Cluster* [Wappler(2007)]. The Test Cluster for a given class can be obtained by performing a transitive static analysis of the signatures of the public methods of this class; each data type (class, interface, or primitive type) encountered during this analysis is added to the Test Cluster. After all methods of the CUT have been included in the Test Cluster, the analysis continues by evaluating all the public methods of the Test Cluster types which have not yet been considered; once all method signatures have been analysed in this manner, the Test Cluster contains all relevant types.

Genetic Programming (GP) is usually associated with the evolution of tree structures; it focuses on automatically creating computer programs by means of evolution [Koza(1992)], and is thus especially suited for representing and evolving Test Programs. GP algorithms maintain a population of candidate solutions, which is iteratively recombined and mutated in order to evolve successive generations of individuals. An individual's probability of being selected for reproduction is associated to its *fitness*, which quantifies the optimality of a solution; the idea is to favour the fitter individuals in the hope of breeding better offspring. Within the tree genomes, the leaf nodes are called *terminals* (and can be inputs to the program, constants or functions with no arguments), whereas the non-leaf nodes are called *non-terminals* (functions taking at least one argument). The *Function Set* is the set of functions from which the GP system can choose when constructing trees. Non-typed GP approaches are, however, unsuitable for representing test programs for OO software, because any element can be a child node in a parse tree for any other element without having conflicting data types; in contrast, *Strongly-Typed Genetic Programming (STGP)* [Montana(1995)] allows the definition of types in the Function Set, which causes the initialization process and the various genetic operations to only construct syntactically correct trees, which can be translated to compilable programs.

Still, syntactically correct and compilable MCSs may still abort prematurely, if a runtime exception is thrown during execution [Wappler and Wegener(2006a)]. Test cases can thus be separated in two classes: *feasible* test cases are effectively executed, and terminate with a call to the MUT; *unfeasible* test cases terminate prematurely because a runtime exception is thrown by an instruction of the MCS.

Several methodologies to the ET of OO software have been proposed, focusing on the usage of distinct EAs – e.g., Genetic Algorithms [Tonella(2004), Inkumsah and Xie(2007), Ferrer et al(2009)], GP [Seesing and Gross(2006)], STGP [Wappler and Wegener(2006a), Wappler and Wegener(2006b), Ribeiro et al(2007), Ribeiro et al(2009)]. However, to the best of our knowledge, there are no studies on the possibility of applying Adaptive EAs to ET problems.

The action of determining the variables and parameters of an EA to suit the problem has been termed *adapting* the algorithm to the problem; in EAs this can be performed dynamically, while the algorithm is searching for a solution. *Adaptive EAs* provide the opportunity to customize the EA to the problem and to modify the configuration and the strategy parameters used while the problem solution is sought. This enables incorporating domain information into the EA more easily, and allows the algorithm itself to select those parametrization which yield better results. Also, these values can be modified during the run of the Adaptive EA to suit the situation during that part of the run. Adaptive EAs have already been applied to solve several problems; interesting review articles include [Angeline(1995), Hinterding et al(1997)]. In [Hinterding et al(1997)], the authors proposed a classification based on the adaptation type and adaptation level of the Adaptive EA. The type of adaptation consists of two main categories: static and dynamic. *Static adaptation* is where the strategy parameters have a constant value throughout the run of the EA; consequently, an external agent or mechanism (e.g., the user) is needed to tune the desired strategy parameters and choose the most appropriate values. *Dynamic adaptation* happens if there is some mechanism which modifies a strategy parameter without external control (e.g., by means of some deterministic rule or some form of feedback from the EA).

3 Adaptive Evolutionary Testing

With STGP approaches, MCSs are encoded (and evolved) as STGP trees; each tree subscribes to a Function Set, which must be specified beforehand and defines the STGP nodes and establishes the constraints involved in the trees' construction. In other words, the Function Set contains the set of instructions from which the algorithm can choose when building the MCSs that compose test cases.

The Function Set can be defined completely automatically based solely on the Test Cluster information [Wappler(2007)]. The definition of the Test Cluster is, therefore, of paramount importance to the algorithm's performance and accuracy. If the Test Cluster consists of many classes (or if it is composed of few classes which possess a high number of public methods), the Function Set can be extremely large. With an increasing size of the Function Set (and hence an increasing size of the search space) the probability that the "right" methods appear in a candidate test sequence decreases – and so does the efficiency of the evolutionary search. Conversely, if a more conservative strategy is employed, the Test Cluster may not include all the classes needed to attain full coverage, thus compromising effectiveness. As such, the selection of the classes and methods to be included in the Test Cluster – and, consequently, in the Function Set – must be carefully pondered, and adequate strategies must be employed for defining the Test Cluster and sampling the search domain.

Still, there are good reasons to suppose that there is no one strategy, however clever, recursive, or self-organizing that will be optimal for all problem domains.

The Test Cluster parametrization process is heavily problem-specific and, as such, it usually falls on the users' hands. Leaving this task to the user has, however, several drawbacks. Namely: the users' mistakes in setting the parameters could be sources of errors and/or suboptimal performance; parameter tuning costs a lot of time; and the optimal parameter value may vary during the evolution [Hinterding et al(1997)].

What's more, the users' choices are inevitably biased, and performance is (arguably) often compromised for the sake of accuracy; in the particular case of ET problems, not doing so could result in the impossibility of obtaining suitable test sets, in conformity to the criteria defined. In [Wappler(2007)], Wappler suggested strategies for addressing the problem of large Function Sets, that result from large Test Clusters with classes that possess many methods:

- Performing a static analysis so as to eliminate all the functions in the Function Set that correspond to methods which are neither object-creating nor state-changing. An Input Domain Reduction strategy, based on the concept of Purity Analysis, that meets this suggestion has already been proposed in [Ribeiro et al(2009)].
- Defining a distance-based heuristic, that prevents the methods from those Test Cluster classes that are associated to the CUT via several other classes from being transformed to functions of the Function Set. Such an heuristic would have to be heavily problem-specific, and decisions would have to be made statically and *a priori* – potentially compromising the success of the search. It seems difficult to implement an automated solution for this idea without compromising generality.
- Naming classes whose methods shall not be transformed to functions of the Function Set. This idea exploits the user's knowledge of the CUT, and suffers from the drawbacks mentioned above.

We propose a different strategy, based on the concept of dynamically adapting the Function Set's constraints selection probabilities. During an evolutionary run, it is possible to perceive that the introduction of certain instructions should be favoured. By allowing the constraints' selection probabilities to fluctuate throughout the search, with basis on the feedback obtained by the behaviour of the individuals produced and evaluated previously, the introduction of interesting genetic material will be promoted. This strategy allows mitigating the negative effects of including a large number of entries into the Test Cluster; also, it allows a higher degree of freedom when defining the Test Cluster, by minimizing the impact of redundant, irrelevant or erroneous choices.

Mutation plays a central role on the diversification of the search and on the exploration of the search space; it basically consists of selecting a mutation point in a tree, and substituting the sub-tree rooted at the point selected with a newly generated sub-tree [Koza(1992)]. Previous studies indicate that better results can be attained if the mutation operator is assigned a relatively high probability of selection [Ribeiro et al(2009)].

Mutation is, in fact, the main process by which new genetic material is introduced during the evolutionary search. In the particular case of ET of OO software problems, it allows the introduction of new sequences of method calls into the generated test cases, so as to allow trying out different objects and states in the search for full

Table 1 Example Function Set and Type Set.

Function Name	**Return Type**	**Child Types**
void print(Object)	TREE	OBJECT
String intToStr(Integer)	STRING (rank:0.2)	INT
"Foo"	STRING (rank:0.8)	
"Bar"	STRING (rank:0.2)	
Integer add(Integer,Integer)	INT (rank:0.8)	INT, INT
0	INT (rank:0.4)	
1	INT (rank:0.6)	
Set Types: OBJECT = [STRING, INT]		

structural coverage. Also, it is clear that during an evolutionary run, it is possible to perceive that some method calls are more relevant than others, e.g. because they had been less prone to throw runtime exceptions and their introduction will likely contribute to test case feasibility, or simply because they have been used less frequently and their introduction will promote diversity (precisely the main task of the Mutation operator).

Whenever mutation occurs, a new (sub-)tree must be created; usually, one of the standard tree builders (e.g., Grow, Full, Half-Builder or Uniform) is used to generate these trees [Luke(2000a)]. We propose employing Luke's Strongly-Typed Probabilistic Tree Creation 2 (PTC2) algorithm [Luke(2000b)] to perform this task, so as to take advantage of the built-in feature that allows assigning probabilities to the selection of constraints. What's more, we have modified this algorithm in order to be able to dynamically update the constraints' probabilities during the evolutionary run.

The Strongly-Typed Probabilistic Tree Creation 2 algorithm works as follows: it picks a random position in the horizon of the tree (i.e., unfilled child node positions), fills it with a non-terminal (thus extending the horizon), and repeats this process until the number of nodes (non-terminals) in the tree, plus the number of unfilled node positions, is greater or equal to the requested tree size. Finally, the remaining horizon is filled with terminals. The tree size is provided by the user.

PTC2 provides uniform distribution of functions and has very low computational complexity [Luke(2000a)]. Also – and most interestingly – PTC2 has provisions for picking non-terminals with a certain probability over other non-terminals of the same return type, and terminals over other terminals likewise. In order to illustrate the methodology followed by this algorithm, let us consider a simple problem which includes a Function Set (Table 1) composed of seven entries (or constraints), defining three non-terminal nodes – `void print(String)`, `String intToStr(Integer)`, `Integer add (Integer, Integer)` – and four terminal nodes – `"Foo"`, `"Bar"`, `0` and `1`. Also, it defines three atomic types – TREE, STRING and INT – and one set type – OBJECT, which includes both INT and STRING. The TREE type is used as return type of the STGP tree.

The constraint selection rankings are also defined. `"Foo"` is given a rank of 0.8, and `"Bar"` a rank of 0.2, for example; this means that, if the PTC2 algorithm is

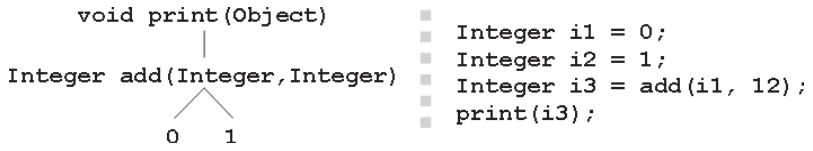

Fig. 1 Example STGP tree *(left)* and corresponding Method Call Sequence *(right)*.

required to select a terminal node with a STRING return type, it will select constraint `"Foo"` with a probability of 80% and `"Bar"` with a probability of 20%. If, however, it is required to select a terminal node with an OBJECT return type, PTC2 uniformly distributes the rankings of the STRING and INT atomic types, with the constraints probabilities being defined as follows: `"Foo"`–40%; `"Bar"`–10%; `0`–20%; `1`–30%. Continuing with this example, if required to grow a tree of size 3, the PTC2 algorithm would build the tree depicted in Figure 1 with a 19.2% chance: 100% probability of selecting the root node, times 80% probability of selecting the non-terminal constraint `Integer add(Integer, Integer)` as an OBJECT type provider for the root node, times 40% chance of choosing `0` as the first terminal of type INT, times 60% chance of selecting `1` as the second terminal.

The dynamic adaptive strategy described in the following Subsection aims at dynamically tuning the Function Set's constraints selection rankings, so as to promote the creation of sub-trees, for insertion in the population via mutation, that favour both feasibility and diversity.

3.1 Dynamic Adaptation Strategy

Let the *constraint selection ranking* of constraint c in generation g be identified as ρ_c^g. Also, let λ be the *runtime exceptions caused factor*, σ be the *runtime exceptions caused by ancestors factor*, and γ be the *constraint diversity factor*. Then, ρ_c^g is updated, at the beginning of each generation, in accordance to the following Equation.

$$\rho_c^g = \rho_c^{g-1} - \lambda_c^{g-1} - \sigma_c^{g-1} - \gamma_c^{g-1} \tag{1}$$

That is, the constraint selection ranking ρ_c^g of a given constraint c in generation g is calculated as being the constraint selection ranking ρ of the previous generation, minus the λ factor of the previous generation (with $\lambda \in [0,1]$), minus the σ factor of the previous generation (with $\sigma \in [0,1]$), minus the γ factor of the previous generation (with $\gamma \in [-1,1]$).

In order to calculate the normalized constraint selection ranking ρ'^g_c, if the minimum ρ_c^g in generation g is negative, the data is firstly shifted by adding all numbers with the absolute of the minimum ρ_c^g; then, ρ'^g_c is normalized into the range of $[0,1]$ as follows.

$$n'^{g}_{c} = \frac{n^{g}_{c}}{n^{g}_{MAX} - n^{g}_{MIN}} \tag{2}$$

The following subsections detail the procedure used for calculating the λ, σ, and γ factors.

Runtime Exceptions Caused Factor Let E^{g}_{c} be the set of runtime exceptions caused by constraint c in generation g, and T^{g} be the set of trees produced in generation g, with $\left|E^{g}_{c}\right|$ and $|T^{g}|$ being their cardinalities. Then, λ is calculated as follows.

$$\lambda^{g}_{c} = \frac{\left|E^{g}_{c}\right|}{|T^{g}|} \tag{3}$$

That is, the λ factor is equal to the number of runtime exceptions thrown by instructions corresponding to constraint c, dividing by the total number of trees. It should be noted that only a single runtime exception may be thrown by MCS (i.e., by tree). This factor's main purpose is that of penalizing the ranking of constraints corresponding to instructions that have caused runtime exceptions to be thrown in the preceding generation. This factor is normalized into the range of $[0,1]$ using Equation 2.

Runtime Exceptions Caused by Ancestors Factor Let X^{g}_{c} be the set of runtime exceptions thrown by ancestors of constraint c in generation g, and $x^{g}_{c^a} \in X^{g}_{c}$ be a runtime exception thrown by an ancestor of level a, with $a \in \{2 = \mathit{parent}, 3 = \mathit{grandparent}, \ldots\}$ being the *ancestry level* of the constraint that threw the exception. Also, let A^{g}_{c} be the multiset containing the ancestry levels of $x^{g}_{c^a} \in X^{g}_{c}$. Then, σ is calculated as follows.

$$\sigma^{g}_{c} = \sum_{a \in A^{g}_{c}} a^{-1} \tag{4}$$

That is, the σ factor is equal to the sum of the inverses of the ancestry levels of the ancestors of constraint c that threw runtime exceptions. This factor's main purpose is that of penalizing the ranking of constraints corresponding to instructions which have participated in the composition of sub-trees (i.e., sub-MCSs) that have caused runtime exceptions to be thrown in the preceding generation; the higher the ancestry level, the lower the penalty. This factor is normalized into the range of $[0,1]$ using Equation 2.

Constraint Diversity Factor Let C^{g} be a multiset containing the number of times each constraint appeared in generation g, and c^{g} be the number of times constraint c appeared in generation g. Also, let m_{C^g} be the mean of the values contained in multiset C^{g}, and $d^{g}_{c} = c^{g} - m_{C^g}$ be the deviation of constraint c in generation g, and $r^{g}_{d} = d^{g}_{MAX} - d^{g}_{MIN}$ be the range of deviation for generation g. Then, γ^{g}_{c} is calculated as follows.

$$\gamma_c^g = \frac{d_c^g}{r_d^g} \tag{5}$$

The γ factor's main purposes are those of allowing constraints to recover their ranking if they have been being used infrequently, and penalizing the ranking of constraints which have been selected too often.

4 Experimental Studies

The adaptive strategy described in the preceeding Section was embedded into *eCrash* [Ribeiro et al(2009)], an automated ET tool, with the objective of observing the impact of this technique on both the efficiency and effectiveness of the test case generation process.

eCrash's approach to ET involves representing and evolving test cases for OO software using the STGP paradigm. The methodology for evaluating the quality of test cases includes instrumenting the MUT, and executing it using the generated test cases – with the intention of collecting trace information with which to derive coverage metrics. The MUTs are represented internally by weighted Control-Flow Graphs (CFGs); the strategy for favouring test cases that exercise problematic structures involves re-evaluating the weight of CFGs' nodes every generation. The aim is that of efficiently guiding the search process towards achieving full structural coverage – i.e., generating a set of test cases that traverse all CFG nodes of the MUT. A thorough description of *eCrash* can be found in [Ribeiro et al(2009)].

The Java `Vector` and `BitSet` classes (JDK 1.4.2) were used as test objects. The rationale for employing these classes is related with the fact that they represent "real-world" problems and, being container classes, possess the interesting property of containing explicit state, which is only controlled through a series of method calls [Arcuri and Yao(2007a)]. Additionally, they have been used in several other case studies described in literature (e.g., [Tonella(2004), Wappler and Wegener(2006b), Arcuri and Yao(2007a)]), providing an adequate test object set in the lack of common benchmark cluster that can be used to test and compare different techniques [Arcuri and Yao(2007b)].

4.1 Setup

The experiments were executed using an Intel Core2 Quad 2.60GHz processor with 4.0 GB RAM desktop, with four test case generation processes running in parallel. 20 runs were executed for each of the 67 MUTs – in a total of 820 runs for the `Vector` class and 520 runs for the `BitSet` Class. Half of these runs were executed employing the adaptive strategy proposed, and half using a "static" approach for comparison purposes. The only difference between the adaptive and the static runs was that, in the latter, the constraints' rankings remained unaltered throughout the

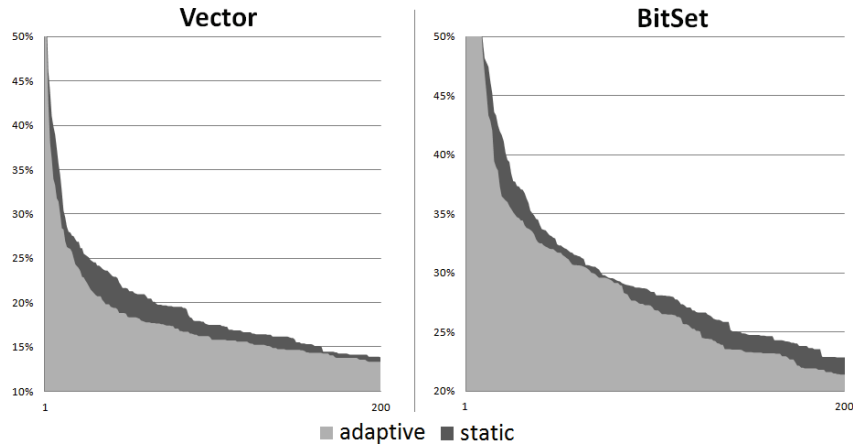

Fig. 2 Average percentage of CFG nodes left to be covered per generation, for `Vector` and `BitSet` classes, with and without adaptation.

evolutionary search. Since the same seeds were used in both the adaptive and non-adaptive runs, and because *eCrash* is deterministic, the discrepancies in the results will solely mirror the impact of the adaptive technique employed.

A single population of 10 individuals was used; the rationale for selecting a relatively small population size had to do with the adaptive algorithm's need of obtaining frequent feedback. The search stopped if an ideal individual was found or after 200 generations. For the generation of individuals, 3 child sources were defined: strongly-typed versions of Mutation (selection probability: 40%) and Crossover (selection probability: 30%), and a simple Reproduction operator (selection probability: 30%). The selection method was Tournament Selection with size 2. The tree builder algorithm was PTC2 (for the reasons explained in Section 3), with the maximum and minimum tree depths being defined as 1 and 4. The constraints' ranking were initialized with the value 1.0, and were updated at the beginning of every generation (before individuals were produced), in accordance to Equation 1.

4.2 Results

Table 2 depicts the percentage of successful runs (i.e., runs in which a test set attaining full structural coverage was found) for the MUTs of the `Vector` and `BitSet` classes, with and without adaptation. The graphs shown in Figure 2 contain the percentage of CFG nodes remaining per generation using the adaptive and the static techniques for the classes tested; their inclusion enables the analysis of the strategy's impact during the course of the search.

The results depicted in Table 2 clearly indicate that the tests case generation process's performance is improved by the inclusion of the Adaptive ET methodology

Table 2 Percentage of runs to attain full structural coverage, for the MUTs of the `Vector` and `BitSet` classes, with and without adaptation.

`Vector`		
MUT	***adaptive***	***static***
void add(int,Object)	80%	**90%**
boolean add(Object)	100%	100%
Object get(int)	100%	100%
int hashCode()	100%	100%
Object clone()	0%	0%
int indexOf(Object)	100%	100%
int indexOf(Object,int)	**20%**	10%
void clear()	100%	100%
boolean equals(Object)	100%	100%
String toString()	100%	100%
boolean contains(Object)	**50%**	40%
boolean isEmpty()	100%	100%
int lastIndexOf(Object,int)	0%	0%
int lastIndexOf(Object)	100%	100%
boolean addAll(Collection)	**90%**	70%
boolean addAll(int,Collection)	**30%**	20%
int size()	100%	100%
Object[] toArray()	100%	100%
Object[] toArray(Object[])	40%	40%
void addElement(Object)	100%	100%
Object elementAt(int)	100%	100%
Object remove(int)	**20%**	0%
boolean remove(Object)	100%	100%
Enumeration elements()	100%	100%
Object set(int,Object)	**100%**	80%
int capacity()	100%	100%
boolean containsAll(Collection)	100%	100%
void copyInto(Object[])	100%	100%
void ensureCapacity(int)	100%	100%
Object firstElement()	100%	100%
void insertElementAt(Object,int)	80%	**90%**
Object lastElement()	100%	100%
boolean removeAll(Collection)	100%	100%
void removeAllElements()	100%	100%
boolean removeElement(Object)	30%	**40%**
void removeElementAt(int)	**20%**	10%
boolean retainAll(Collection)	100%	100%
void setElementAt(Object,int)	**100%**	70%
void setSize(int)	100%	100%
List subList(int,int)	**30%**	0%
void trimToSize()	100%	100%

`BitSet`		
MUT	***adaptive***	***static***
boolean get(int)	**90%**	60%
BitSet get(int,int)	0%	0%
int hashCode()	100%	100%
Object clone()	0%	0%
void clear(int, int)	0%	0%
void clear()	100%	100%
void clear(int)	**90%**	80%
boolean equals(Object)	0%	0%
String toString()	100%	100%
boolean isEmpty()	100%	100%
int length()	**30%**	0%
int size()	100%	100%
void set(int)	**100%**	70%
void set(int, boolean)	100%	100%
void set(int, int)	**70%**	40%
void set(int, int, boolean)	40%	**70%**
void flip(int, int)	**60%**	20%
void flip(int)	**90%**	50%
void and(BitSet)	0%	0%
void andNot(BitSet)	**60%**	30%
int cardinality()	100%	100%
boolean intersects(BitSet)	**20%**	0%
int nextClearBit(int)	0%	0%
int nextSetBit(int)	10%	10%
void or(BitSet)	0%	0%
void xor(BitSet)	**90%**	30%

proposed. The adaptive strategy outperformed the static approach for 28.4% of the MUTs tested, whereas the latter only surpassed the former in 5.9% of the situations. In terms of the average success rate, the adaptive strategy enhances results by 3% for the `Vector` class; the improvement is even more significant for the `BitSet` class, with the results meliorating 11%. What's more, the adaptive strategy allowed attaining full structural coverage in some situation in which the success rate had been of 0% using the non-adaptive strategy – namely, for the `Object remove(int)` and `List subList(int,int)` MUTs of the `Vector` class, and for the `int length()` and `boolean intersects(BitSet)` MUTs of the `BitSet` class; these observations indicate that this strategy is specially suited for overcoming some difficult state problems.

The graph shown in Figure 2 also provides clear indication that the evolutionary search benefits from the inclusion of the adaptive approach described. For the `Vector`'s MUTs, the average number of nodes remaining when the Adaptive ET approach is used decreases as much as 6% during the initial generations; for the `BitSet` class, the contrast is the results is less perceptible, but the adaptive approach still manages to attain a 3% improvement at certain stages.

In terms of speed, the overhead introduced by embedding the adaptive strategy into the evolutionary algorithm was negligible; each generation took, on average, 23.25 seconds using the adaptive methodology, and 23.21 seconds using the static approach. The time overhead introduced by the adaptation procedure was a mere 0.19%.

5 Conclusions

Recent research on Evolutionary Testing has relied heavily on Genetic Programming for representing and evolving test data for Object-Oriented software. The main contribution of this work is that of proposing a dynamic adaptation strategy for promoting the introduction of relevant Method Call Sequences into the generated test cases by means of Mutation.

The Adaptive Evolutionary Testing strategy proposed obtains feedback from the individuals produced and evaluated in the preceding generation, and dynamically updates the selection probability of the constraints defined in the Function Set so as to encourage the selection of interesting genetic material and promote diversity and test case feasibility. The experimental studies implemented indicate a considerable improvement in the algorithm's efficiency when compared to its static counterpart, while introducing a negligible overhead.

References

[Angeline(1995)] Angeline PJ (1995) Adaptive and self-adaptive evolutionary computations. In: Computational Intelligence: A Dynamic Systems Perspective, IEEE Press, pp 152–163

[Arcuri and Yao(2007a)] Arcuri A, Yao X (2007a) A memetic algorithm for test data generation of object-oriented software. In: Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC), IEEE, pp 2048–2055

[Arcuri and Yao(2007b)] Arcuri A, Yao X (2007b) On test data generation of object-oriented software. In: TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, IEEE Computer Society, Washington, DC, USA, pp 72–76

[Ferrer et al(2009)] Ferrer J, Chicano F, Alba E (2009) Dealing with inheritance in oo evolutionary testing. In: GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, ACM, New York, NY, USA, pp 1665–1672

[Hinterding et al(1997)] Hinterding R, Michalewicz Z, Eiben AE (1997) Adaptation in evolutionary computation: A survey. In: Proceedings of the Fourth International Conference on Evolutionary Computation (ICEC 97), IEEE Press, pp 65–69

[Inkumsah and Xie(2007)] Inkumsah K, Xie T (2007) Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In: Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), pp 425–428

[Koza(1992)] Koza JR (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). The MIT Press

[Luke(2000a)] Luke S (2000a) Issues in scaling genetic programming: Breeding strategies, tree generation, and code bloat. PhD thesis, Department of Computer Science, University of Maryland, A. V. Williams Building, University of Maryland, College Park, MD 20742 USA

[Luke(2000b)] Luke S (2000b) Two fast tree-creation algorithms for genetic programming. IEEE Transactions on Evolutionary Computation 4(3):274–283

[McMinn(2004)] McMinn P (2004) Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14(2):105–156

[Montana(1995)] Montana DJ (1995) Strongly typed genetic programming. Evolutionary Computation 3(2):199–230

[Ribeiro et al(2007)] Ribeiro JCB, Zenha-Rela M, de Vega FF (2007) An evolutionary approach for performing structural unit-testing on third-party object-oriented java software. In: Krasnogor N, Nicosia G, Pavone M, Pelta DA (eds) NICSO, Studies in Computational Intelligence, vol 129, Springer, pp 379–388

[Ribeiro et al(2009)] Ribeiro JCB, Zenha-Rela M, de Vega FF (2009) Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. Information & Software Technology 51(11):1534–1548

[Seesing and Gross(2006)] Seesing A, Gross HG (2006) A genetic programming approach to automated test generation for object-oriented software. ITSSA 1(2):127–134

[Tonella(2004)] Tonella P (2004) Evolutionary testing of classes. In: ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, ACM Press, New York, NY, USA, pp 119–128

[Wappler(2007)] Wappler S (2007) Automatic generation of object-oriented unit tests using genetic programming. PhD thesis, Technischen Universitat Berlin

[Wappler and Wegener(2006a)] Wappler S, Wegener J (2006a) Evolutionary Unit Testing Of Object-Oriented Software Using A Hybrid Evolutionary Algorithm. In: CEC'06: Proceedings of the 2006 IEEE Congress on Evolutionary Computation, IEEE, pp 851–858

[Wappler and Wegener(2006b)] Wappler S, Wegener J (2006b) Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In: GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM Press, New York, NY, USA, pp 1925–1932