It is Not the Length that Matters, It is How You Control It

Gordon Fraser
Saarland University – Computer Science
Saarbrücken, Germany
fraser@cs.uni-saarland.de

Andrea Arcuri
Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

***Abstract*—The length of test cases is a little investigated topic in search-based test generation for object oriented software, where test cases are sequences of method calls. While intuitively longer tests can achieve higher overall code coverage, there is always the threat of *bloat* – a complex phenomenon in evolutionary computation, where the length abnormally grows over time. In this paper, we show that bloat indeed also occurs in the context of test generation for object oriented software. We present different techniques to overcome the problem of length bloat, and evaluate all possible combinations of these techniques using different search lengths. Experiments on a set of difficult search targets selected from several open source and industrial projects show that the important choice in search-based testing is not the length of test cases, but how to make sure that this length does not become bloated.**

***Keywords*-test case generation; search-based testing; test case length; bloat control**

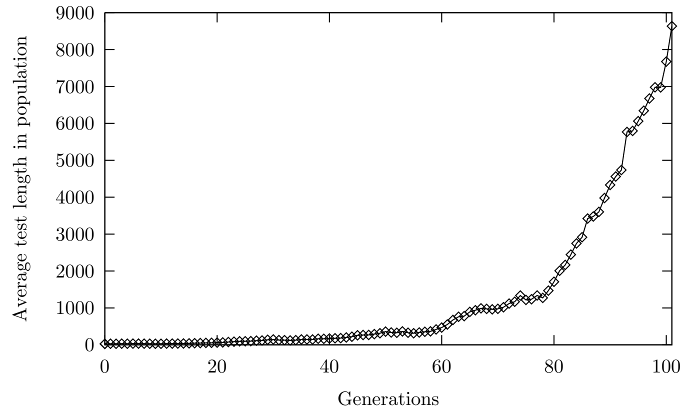


Fig. 1. Bloat occurring during the search for a test case to cover a branch of the `XMLElement` class in NanoXML. As the evolution progresses, the average length of the population increases exponentially.

I. Introduction

Deriving test cases for object oriented software entails generation of sequences of method calls. Search-based techniques have been demonstrated to be a suitable tool for this task [1], [2], but raise important questions such as the choice of a search *length* for these method sequences. The length, however, is not only an important parameter of the search but also one of its biggest threats: *Bloat* is a phenomenon in evolutionary search where the length of individuals increases to the point of making the search impossible.

For example, consider Figure 1, which shows the average length of the test cases in a population of a genetic algorithm. As a typical example of test case generation, the aim of this search is to find a sequence of method calls that will cover a non-trivial branch of the `XMLElement` class in the open source Java project NanoXML. Without any techniques to control bloat, the test cases become longer and longer after each generation of the search, until all the memory is consumed.

Bloat is an extremely complex phenomenon in evolutionary computation, and after many decades of research it is still an open problem whose dynamics and nature are not completely understood [3]. Unfortunately, in the past the issue of length has largely been neglected in the context of test case generation, and so there is no conclusive evidence on what length to choose and how to prevent it from being bloated.

In this paper, we analyze the effects of length and bloat in the context of testing object oriented software. The contributions of this paper are as follows:

Bloat: We propose and evaluate a set of different techniques to control bloat, identifying which combinations of techniques work best and should therefore be used in the future.

Length: We analyze the effect of the test case length on the results and on bloat, showing that the length has only small influence on both, resulting test suites and bloat.

The evaluation of this paper considers a set of 100 difficult branches selected from six open source projects and an industrial case study, and experiments are performed on 96 different configurations and three different lengths, repeated with 25 random seeds each, resulting in a significant amount of data backing our results.

This paper is organized as follows: First, we give an overview of the bloat and length problems as well as previous work in Section II. In Section III we instantiate the concrete domain for our experiments: testing of object oriented software. Section IV describes different techniques that can be applied to control bloat. Section V describes the experiments and discusses the results in detail. Finally, Section VI discusses threats to the validity of our study, and Section VII concludes the paper.

II. Background

As the number of possible test cases is usually infinite, a practical solution is to choose a coverage criterion, which represents a finite set of coverage goals. The objective is to obtain a test suite that, once executed, covers as many as possible of these goals. Unfortunately, for non-trivial software, writing such test suites by hand is a complex and tedious task. Therefore, automated techniques have been designed to address this task. The predominant criterion in the literature on structural testing is branch coverage, but in principle any other coverage criterion (e.g., mutation testing [4]) is amenable to automated test generation.

For some testing goals it can be easy to find test input data to cover them, but for other goals it might be very difficult to find such data. Therefore, a common approach is to use a first step of random testing to cover the easy branches [5]. After this initial phase, there is a second phase in which more sophisticated techniques are used to target all the remaining uncovered goals. A common approach in the literature is to target one such goal at a time, generating test inputs either symbolically [6] or with a search-based approach [1]. In this paper, we focus our analyses on this second phase: finding test data to cover difficult to reach testing goals, in particular for branch coverage.

Meta-heuristic search techniques have been suggested as a possible solution to automate test case generation [1], [2]. In the context of object oriented software, test cases are essentially small programs exercising the classes under test. Search-based techniques have been applied to test object oriented software using method sequences [7], [8] and strongly typed genetic programming [9], [10]. A promising avenue seems to be the combination of evolutionary methods with dynamic symbolic execution (e.g., [11]), alleviating some of the problems both approaches have.

While we aim to obtain the highest achievable coverage, it is important that the resulting test suites should be *small*. In fact, in this paper we assume the general case in which no *automated oracle* is available. In such a case, the output of each test case needs to be manually checked (e.g., by writing appropriate assert statements). This is often the case in unit testing. Therefore, it is not feasible to ask a software tester to manually write assert statements for thousands of test cases. Long test sequences are also more difficult to analyze and to understand. For all these reasons, there has been work in which the goal was still obtaining highest coverage of the desired testing criterion, but with the secondary goal of obtaining a test suite that is as small as possible (e.g., [12]–[14]).

Effectively, this means that there are two conflicting goals: maximizing coverage C while minimizing the size of the test suite S. How to combine these two measures? An approach would be to use a pareto-based multi-objective algorithm [15], but the problem is that the length/size is less important than coverage. Arcuri and Yao [12] used the following fitness function to maximize coverage: $C + (1/S + 1)$, where C is coverage and S is the size. In this way, in a comparison between two test cases, better coverage is always preferred regardless of length. On the other hand, Andrews *et al.* [14] used $(C * 1000) - S$, which means that an increase of one point in coverage is better only if it does not result in a test case that is 1000 function calls longer. Baresi *et al.* [13] included the length of test sequences in the fitness function as well, but they do not specify how this was done. Notice that these approaches try to find test sequences that cover as many testing goals as possible. This can lead to potential problems if there are conflicting goals, such that a single sequence cannot cover all goals at once. Another common approach that does not suffer of such a problem is to target one coverage goal at a time [1], [7], [8], [16].

Arcuri [17] studied what is the role of test sequence length on branch coverage. In that work, only container classes were used as case study. Using *longer* sequences made the testing of these container classes trivial even with naive techniques such as random testing. A simple post processing was very effective to minimize such sequences without compromising their coverage.

There has been other related work to shed light on the role of length of test sequences. Andrews *et al.* [18] studied whether for the *fault detection* of random testing it is better to have few long sequences or many short ones. Similar work has been carried out by Fraser and Gargantini [19].

III. Evolutionary Testing of Object Oriented Software

Search-based testing uses meta-heuristic search techniques to evolve an initial set of candidate test cases towards satisfying a given test objective, for example to reach a certain branch in the control flow of the software under test. In this section, we describe the techniques commonly used in search-based testing for object oriented software, which are also those used for experimentation in this paper.

A. Genetic Algorithms

A genetic algorithm is a meta-heuristic search technique that tries to imitate the mechanisms of natural adaptation by evolving a population of candidate solutions using genetics-inspired operations. Algorithm 1 shows a commonly used version of such a genetic algorithm, a steady state genetic algorithm: Starting with a randomly generated initial population, parents are selected using, for example, rank selection [20], and then crossed over and mutated with a certain probability. Depending on the fitness values, either the offspring or the parents are carried over to the next population. An iteration is done if the next generation has reached the same size as the current generation. This process is repeated until either an optimal solution has been found, or some other criterion stops the search (e.g., maximum allowed resources spent).

B. Fitness Function

The fitness function of a test case generation search depends on the chosen coverage criterion. In this paper, we consider branch coverage, which is also the predominant criterion used

Algorithm 1 A steady state genetic algorithm as used for search-based testing.

```
1  current_population ← generate random population
2  repeat
3    Z ← elite of current_population
4    while |Z| ≠ |current_population| do
5      P1,P2 ← select two parents with rank selection
6      if crossover probability then
7        O1,O2 ← crossover P1,P2
8      else
9        O1,O2 ← P1,P2
10     mutate O1 and O2
11     f_P = min(fitness(P1),fitness(P2))
12     f_O = min(fitness(O1),fitness(O2))
13     if f_O ≤ f_P then
14       Z ← Z ∪ {O1,O2}
15     else
16       Z ← Z ∪ {P1,P2}
17   current_population ← Z
18 until solution found or maximum resources spent
```

in the literature and in practice. A traditional fitness function for branch coverage [1], [2] consists of the *approach level* and the *branch distance*:

The *approach level* A is used to guide the search towards the target branch. It is determined as the minimal number of control dependent edges in the control dependency graph between the target branch and the control flow represented by the test case.

The *branch distance* B is a common heuristic to guide the search for input data to solve the constraints in the logical predicates of the branches [1]. The branch distance for any given execution of a predicate can be calculated by applying a recursively defined set of rules (see [1] for details). For example, for predicate $x \geq 10$ and $x = 5$, the branch distance to the true branch is $10 - 5 + k$, with $k \geq 1$. In practice, to determine the branch distance, each predicate of the software under test is instrumented to evaluate and keep track of the distances for each execution.

To avoid that the branch distance dominates the approach level, the branch distance has to be normalized in $[0,1]$. A recommendable normalizing function $\nu(x)$ is that suggested by Arcuri [21]: $\nu(x) = x/(x+1)$. The fitness function for test case t and branch coverage goal c can therefore be defined as follows:

$$\text{fitness}(t,c) = A_c + \nu(B_c) \quad (1)$$

C. Problem Representation

In search-based testing for object oriented software, test cases and thus also the chromosomes of the genetic algorithm are sequences of statements [7], [8]. A statement can be a call to a constructor, a method call, or a reference to a field or primitive value. Parameters of method and constructor calls, and source objects of method calls and field accesses have to be objects generated in the same test case at a previous position.

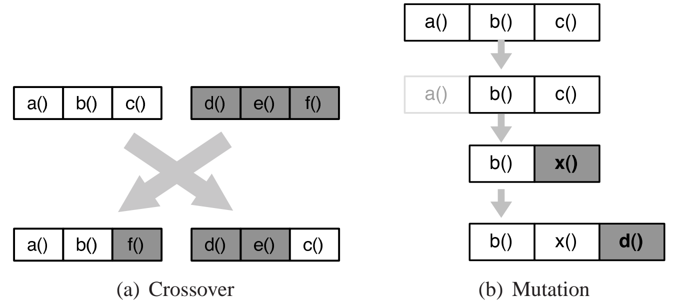


Fig. 2. Crossover and mutation are the basic operators for the search using a GA.

Primitive statements represent numeric variables, e.g., `int var0 = 54`.

Constructor statements generate new instances of any given class; e.g., `XMLElement var1 = new XMLElement()`.

Field statements access public member variables of objects, e.g., `int var2 = var1.line_nr`.

Method statements invoke methods on objects or call static methods, e.g., `int var3 = var1.countChildren()`.

A test case is a sequence of such statements, and the length of a test case is the number of statements it consists of.

D. Crossover

Crossover creates two offspring test cases from two parent test cases P_1, P_2. Different flavors of crossover operators have been defined; in evolutionary testing of classes usually a single point crossover is used, meaning that each of the parent chromosomes is split at a single point, and the constituent parts of the parents are merged together (see Figure 2(a)).

Crossover functions can further vary in how the crossover point is chosen. Tonella [7] chooses a random point in the range of $[1, min(length(P_1), length(P_2))]$. Baresi *et al.* [13] and Fraser and Zeller [8] choose different random positions for each of the parents in the range $[1, length(P_1)]$ and $[1, length(P_2)]$. In this paper, we call this latter crossover operator *Two Point Crossover* (TPX).

As statements in the test cases might have dependencies, it is necessary to try to satisfy these dependencies when attaching two sub-sequences from the parents. If there are alternative objects of the required type, then one of these objects is randomly chosen. If there is no object that would satisfy the dependency, then additional statements need to be added to create the required object.

E. Mutation

Mutation introduces local changes into individuals. When applying mutation to sequences of method calls, we distinguish three main types of mutation, illustrated in Figure 2(b):

Deletion: This mutation operator removes a statement from a test case. As there are dependencies between statements (e.g., a return value might be used as a parameter in another method call), the dependencies need to be resolved, either by recursively deleting dependent statements, or by replacing references with different suitable objects. In a chromosome of length l, each statement is deleted with probability $1/l$.

Change: This mutation operator alters a given statement. For example, Tonella [7] lists a number of different possibilities to change statements. In our experiments, a change replaces a method call with a randomly chosen method call that has the same return type and has all dependencies satisfied at the given position in the test case. Primitive values (e.g., integer numbers) are changed by a random but bounded increase or decrease. In a chromosome of length l, each statement is changed with probability $1/l$.

Insertion: In terms of bloat analysis insertion is the most interesting operator, as it is the only mutation operator that contributes to growth of the length. We use the following strategy to insert statements: With probability σ', a new randomly chosen statement is inserted at a random position in the test case. If it is added, then a second statement is added with probability σ'^2, and so on until the ith statement is not inserted. Parameters of new method calls are either satisfied with existing objects, or lead to addition of further statements to create necessary objects.

To generate the initial population of the search, we sample test cases at random. First, we choose a value r in $1 \leq r \leq W$ with uniform probability, where W is a value that needs to be set (e.g., $W = 80$). Then, on an empty sequence we repeatedly apply the insertion operator described above until the test case has a length $\geq r$. Because on average we expect $r = W/2$, the value of W should not be set too high, otherwise there is the risk of consuming all the available memory.

F. Generating Test Suites

Any non-trivial class will have a number of different coverage goals, even for simple coverage criteria. As discussed in Section II, it is common practice to have a first phase of random testing to cover the easy branches. Then, each remaining target can be individually sought with more sophisticated techniques. Some of these remaining coverage goals may be *infeasible*, which means that there exists no test case that would cover them. To avoid that all available resources are wasted on infeasible or difficult coverage goals on which the search fails, it is necessary to limit the resources spent on a single coverage goal. For this, we apply the following strategy:

- For $|B|$ branches to cover and an initial budget of X statements (or fitness evaluations, generations, etc.), the execution limit for the search on each branch is $X/|B|$.
- If a branch is covered, some budget may be left over, and so after the first iteration on all branches there is a remaining budget X'. For the remaining uncovered branches B' a new budget $X'/|B'|$ is calculated and a new iteration is started on these branches.
- This process is continued until the maximum number of statements is reached.

Test cases are only generated for branches that have not been covered previously by other test cases, as a test case can cover more than one branch.

Algorithm 2 Adapted genetic algorithm that includes bloat checks, highlighted with gray background.

```
1  current_population ← generate random population
2  repeat
3     Z ← elite of current_population
4     while |Z| ≠ |current_population| do
5        P1,P2 ← select with [extended rank selection]
6        if crossover probability then
7           O1,O2 ← [crossover P1,P2 with RPX]
8        else
9           O1,O2 ← P1,P2
10       [mutate O1 and O2 with size check]
11       fP = min(fitness(P1),fitness(P2))
12       fO = min(fitness(O1),fitness(O2))
13       lP = length(P1) + length(P2)
14       lO = length(O1) + length(O2)
15       TB = best individual of current_population
16       if fO < fP [∨(fO = fP ∧ lO ≤ lP)] then
17          for O in {O1,O2} do
18             if [length(O) ≤ 2 × length(TB)] then
19                Z ← Z ∪ {O}
20             else
21                Z ← Z ∪ {P1 or P2}
22       else
23          Z ← Z ∪ {P1,P2}
24    current_population ← Z
25 until solution found or maximum resources spent
```

IV. Bloat Control Techniques

Bloat occurs when small negligible improvements in the fitness value are obtained with larger solutions. This is very typical in classification/regression problems. When in software testing the fitness function is just the obtained coverage, then we would not expect bloat, because the fitness would assume only few possible values. However, when other metrics are introduced with large domains of possible values (e.g., branch distance [1] or mutation impact [8]), then bloat might occur.

Bloat can be a particularly harmful phenomenon. Longer sequences can consume large amounts of memory and take longer to evaluate, which would lead to less generations in the evolutionary search (within the same amount of time). Furthermore, very long sequences cannot be directly used for testing purposes unless an automated oracle is available, which is usually not the case.

The problem of bloat is long known in the field of evolutionary computation, and so different techniques have been

proposed to keep bloat under control. However, there is a difference between the length of the test cases that are given as output after the search and those that exist during the search itself. On one hand, the output sequences should be as short as possible (while optimizing coverage). On the other hand, during the search it can be very useful to have longer sequences [17], because it would make the search able to explore larger areas of the search landscape without being trapped in fitness plateaus. Once the search for maximum coverage is finished, a post-processing can be used to easily remove the unnecessary function calls.

Therefore, the dynamics of bloat and the methods coming from the literature of genetic programming (GP) to contrast it might not behave in a similar manner in the case of testing object-oriented software. This section describes bloat control techniques that can be applied to test case generation.

A. Relative Position Crossover

One possible source of bloat is the crossover function, in which one of the offspring can grow in size. When using TPX we choose two different splitting points in the parents (e.g., P_1 and P_2) at random, then the length of the offspring can be very unbalanced when the splitting points are at the opposite edges of the chromosomes. The length of the offspring would vary between 0 and $length(P_1) + length(P_2)$, with average value $(length(P_1) + length(P_2))/2$.

Another version of the crossover operator generates two offspring O_1 and O_2 from two parent test cases P_1 and P_2. A random value α is chosen from [0,1]. On one hand, the first offspring O_1 will contain the first $\alpha|P_1|$ test cases from the first parent, followed by the last $(1 - \alpha)|P_2|$ test cases from the second parent. On the other hand, the second offspring O_2 will contain the first $\alpha|P_2|$ test cases from the second parent, followed by the last $(1-\alpha)|P_1|$ test cases from the first parent. In this paper, we call this operator *Relative Position Crossover* (RPX), and it is shown in Line 7 in Algorithm 2. In contrast to TPX, in RPX the offspring will never be longer than the longest of the parents.

Regardless of the crossover operator, test cases can still grow in size, as additional statements might be added to satisfy dependencies of merged parts of the parents. In addition, test cases can grow as part of the mutation operator.

B. Fixed Maximum Length

A very common approach to contrast bloat is to put an upper limit L to the length of the test cases, e.g., $L = 100$ function calls. This constraint can be enforced in several ways: First, by having search operators that do not sample offspring that are longer than L (Line 10 in Algorithm 2). For example, an insertion mutation could be avoided if the length already equals L. Second, offspring that are longer than L (e.g., when we use TPX) can be rejected, and the parents will be copied to the next generation instead of the offspring. Finally, the limit can be given implicitly by specifying a maximum amount of resources to be spent per individual. For example, one can define a timeout for the execution of test cases.

But how to choose a maximum length L? Should it be equal to 100 or 1,000? A too small value might make the search very unlikely to succeed. With a large value there might be the risk of running out of memory and being severely affected by bloat. In GP, a rule of thumb is to have trees of maximum depth equal 17. In the case of testing object-oriented software, we are aware of no work that tries to analyze and give an answer to this research problem.

As previously discussed, bloat is a very complex phenomenon. This is illustrated by the fact that, counterintuitively, using a limit L might actually favor the raise of bloat. As a detailed discussion of this would go beyond the scope of this paper, we refer the interested reader to the literature [3].

C. Dynamic Upper Bounds

Choosing a proper value for the upper limit L might not be easy, and there might be side-effects due to the use of a fixed L. Beside L, one further approach discussed by Silva and Costa [3] is to use a dynamic limit based on the best individual T_B in the current generation. For example, an offspring O could be rejected if $length(O) > 2 \times length(T_B)$ (Line 18 in Algorithm 2). In this way, we would not need the burden of fixing a value for L, and would allow a less constrained search of the solution space. For example, if the current best solution has length 10, we would still able to explore sequence up to length 20. Notice that such a dynamic limit can be used in conjunction with the static limit L.

D. Integrating Length in the Ranking Function

Another approach to prevent bloat is to penalize the length directly in the fitness function [8], [12]–[14]. However, as discussed in Section II, combining two different objectives that have different order of measure is not easy. Furthermore, because the branch distance might obtain any possible continuous value, it would not be possible to combine it with the length such that the length would be less important. The fitness function $C + (1/S + 1)$ (where S is the size of the test case) discussed in Section II works only because the coverage C only assumes integer values.

Instead of combining the length in the fitness function in Equation 1, we use a different approach: In general, the fitness function is only used to select individuals for reproduction. In this paper, we use *rank selection* [20] (see Line 5 in Algorithm 2). Test cases are ranked based on their fitness value. Individuals with better fitness will receive a better rank, and so will have higher chances of being selected for reproduction. To penalize longer test sequences without penalizing a better fitness branch distance and approach level, in case of ties in the ranking, we resolve the ties by giving a better rank to the test cases that are shorter.

E. Length Dependent Parent Replacement

The last method we investigate to contrast bloat is based on the relations between the performance of the parents and its offspring. If one offspring has a fitness value strictly better than the fittest of its parents, then both offspring will be accepted

in the new generation independently of their length (but other bloat control methods might still prevent it). However, in case of equal fitness, the offspring will be accepted only if they are not longer than their parents; see Line 16 in Algorithm 2. In other words, we accept longer test sequences in the new generations if and only if at least one of the offspring has strictly better fitness value than both the parents.

V. Experimental Evaluation

To study the effects of both the test case length and the bloat control techniques, we performed a set of experiments. In detail, this evaluation aims to answer the following research questions:

- **RQ1**: How does the maximum starting length W influence the search results?
- **RQ2**: How do the bloat control methods impact the achieved coverage?
- **RQ3**: Which techniques to control bloat are most effective?

A. Case Study

As subject for our experiments, we selected a set of open source Java libraries: Java Collections (a subset of the java.util library), Apache Commons Collections and Commons Primitives, NanoXML, and a Java translation of the String case study subjects used by Alshraideh and Bottaci [22]. We also use a set of numerical applications used in [23] and a subset of classes from an industrial application [24]. This resulted in a large and variegated case study.

This case study resulted in nearly 1,000 classes and more than 15,000 branches— far too many for an in-depth analysis of bloat control methods. We needed a way to filter out the *easy* branches, and identify the difficult ones. This is also of practical interest: If a testing technique $\mathcal{A}$ is twice as fast as another technique $\mathcal{B}$, then solving an easy problem in one millisecond instead of two milliseconds would be an improvement of no value from a practical stand point. On the other hand, solving a problem in one hour instead of two hours would be of practical interest.

In our case study, we applied the following filtering phase to choose a selection of difficult branches: We applied our test case generation tool with a search limit of 200,000 statements per branch (of which there were more than 15,000) with all bloat control techniques enabled, collecting information for each branch about the number of statements executed until a solution was found (one run per branch). Given this information, we selected the subset of those branches which resulted in a solution (i.e., are feasible), but required between 100,000 and 200,000 statements for this solution (i.e., are non-trivial). This resulted in a set of exactly 100 difficult but feasible branches, which we used for further experimentation.

B. Experimental Setup

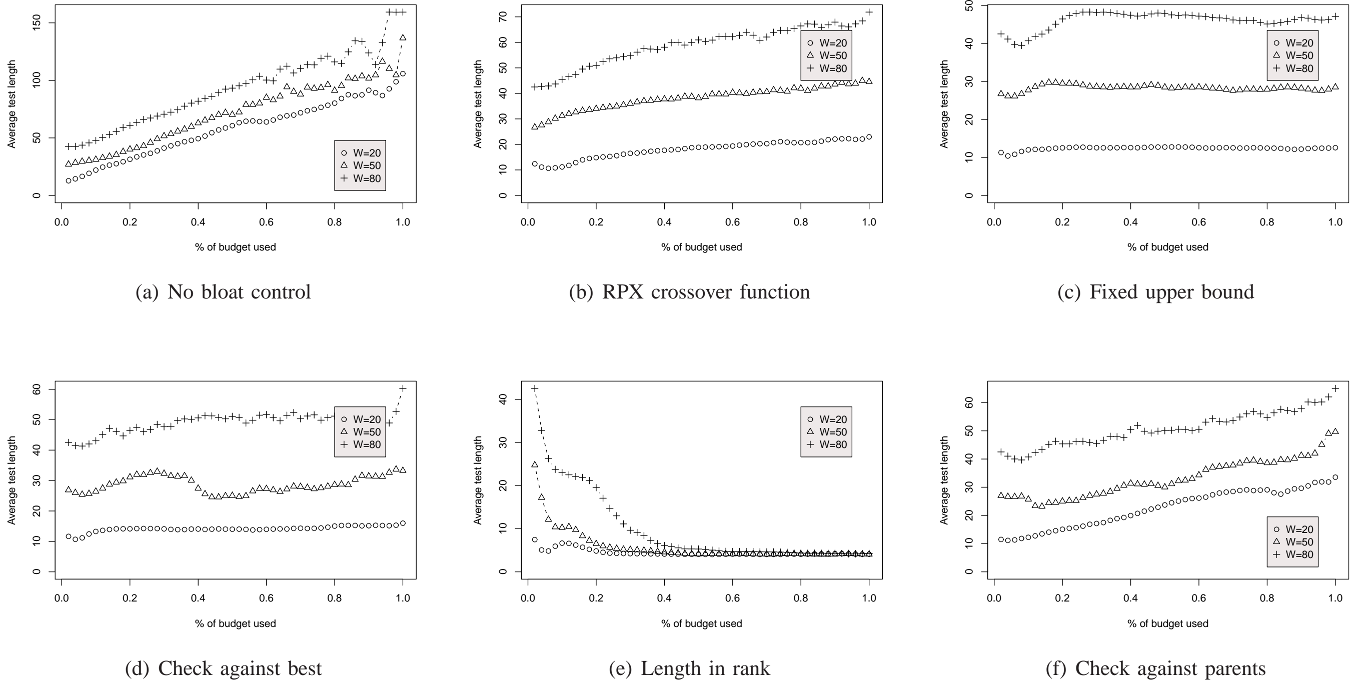
For the experiments we consider the five bloat control techniques described Section IV. In particular, we use the following labels to indicate whether a bloat technique is employed:

- **Bo**: the maximum length for the test cases is bounded from above, i.e., if we set an upper limit L. In particular, we chose $L = W$, where $[1,W]$ is the range in which the length of new random test cases is chosen from.
- **Xo**: the crossover RPX is used instead of TPX.
- **Ra**: use the length of the test cases to resolve the ties in the rank selection of the parents for reproduction.
- **Pa**: check length of offspring against parents' length.
- **Be**: check the length of offspring against best solution's length in the current population.

For the initial length of random test cases, we consider three values for W, specifically $W \in \{20,50,80\}$. For the experiments in this paper, the total number of configurations for the genetic algorithm is hence $2^5 \times 3 = 96$. Because we run the search on each branch independently, this means a total of $96 \times 100 = 9{,}600$ different experiments. In all the experiments, we give a budget of $100{,}000$ statement evaluations (a typical value in the literature, e.g. [16]). The search can finish for two reasons: either a test case that covered the target branch is found (a so called *global optimum*), or the entire execution budget has been consumed.

To compare whether a configuration $\mathcal{A}$ is better than another configuration $\mathcal{B}$ on a branch, we follow the follow procedure, as described in more detail by Arcuri and Briand [25]. We run the genetic algorithm n times for both configurations on that branch (so $2n$ runs), and we record the number of times a out of n an optimal solution is found with the first configuration $\mathcal{A}$, and the number of times b it is found with the other configuration $\mathcal{B}$. The *success rate* for $\mathcal{A}$ is defined as a/n. If $a > b$, then it would seem that $\mathcal{A}$ is better then $\mathcal{B}$, and the other way round if $a < b$. However, because genetic algorithms are randomized, we need rigorous statistical tests to assess whether there is enough empirical evidence to claim with high confidence that the two success rates are indeed different. We apply a Fisher exact test at significance level $\alpha = 0.05$. If the p-value is above the chosen α level, then there would not be enough evidence to claim a difference in the success rates of $\mathcal{A}$ and $\mathcal{B}$. Still, the performance of the two algorithms can be statistically different, as we will now discuss in more detail.

In case there is no statistical difference in the success rates, we can analyze the *time* an algorithm takes to find an optimal solution for the runs in which it is successful [25]. For example, assume that $a = b = n$, i.e., for the given budget of statements the algorithm finds an optimal solution in all the $2n$ runs. This would happen if the target branch is easy and/or the given computational budget is very high. In these cases, we might want to know how fast the algorithm finds a solution. This is of practical importance, because we can stop the search as soon as we find an optimal solution. For each run that leads to find an optimal solution, we can monitor how much computational effort has been spent, measured in the number of statements executed before finding the optimal solution for each run. We can hence compare the computational effort of $\mathcal{A}$ (based on a observations/values) with the effort of $\mathcal{B}$ (based on



(a) No bloat control

(b) RPX crossover function

(c) Fixed upper bound

(d) Check against best

(e) Length in rank

(f) Check against parents

Fig. 3. Evolution for the same branch as in Figure 1, using the different bloat control techniques one at a time. Evolution is bounded by 100,000 executed statements for these graphs, results are averaged for 25 runs.

b observations/values). As discussed in [25], we use a Mann-Whitney U-test (with $\alpha = 0.05$) to asses which configuration requires less computational effort to find optimal solutions.

The genetic algorithm was configured with a population size of 100, and a rank bias of 1.7. The crossover probability was set to 0.75, and mutation is applied with probability $1/3$ each for insertion, deletion, and change. The initial insertion probability σ was set to 0.5. These settings are in line with common practice in the literature and our past experience with genetic algorithms.

C. *Bloat Control Techniques Illustrated*

To illustrate the effects of the individual bloat control techniques, we performed a set of experiments on the branch used to generate the plot in Figure 1. We generated test cases for this branch using 25 different random seeds and a maximum of 100,000 statements, and averaged the results. Figure 3(a) shows the behavior of the length without any bloat control techniques activated—the length grows, as expected. Figure 3(b) shows how the average test case length behaves over the evolution of test generation when we use RPX for the same branch. The average size of the test cases increases, but at a much slower rate than with TPX. Figure 3(c) shows how the average test case length converges when a fixed maximum length is used. Figure 3(d) shows how the average test case length first shrinks as the long individuals of the initial population are removed, and then grows only slowly. Figure 3(e) shows how the use of length in the rank reduces the average test case length for the usual example branch. Finally, Figure 3(f) shows how the average length increases slowly when using the parent check.

D. *Analysis of Individual Bloat Control Techniques*

To study the effects of the individual bloat control techniques in detail, we ran a first set of experiments in which, for each $W \in \{20{,}50{,}80\}$, we ran a genetic algorithm with no bloat control activated (**No**) and with the five bloat control activated one at a time, for a total of $3 \times (1 + 5) = 18$ configurations for each branch (i.e., a subset of the total 96 configurations). This first set of experiments is used to assess the implication of each bloat control method in isolation. In fact, the case of multiple combinations of bloat control methods is harder to analyze and visualize.

For each configuration and for each branch, we ran the genetic algorithm $n = 25$ times, for a total of $100 \times 18 \times 25 = 45{,}000$ runs. Figure 4 shows 18 boxplots, one for each analyzed configuration. Each boxplot shows the distribution of success rates on the 100 branches for that configuration. Table I and II summarize the statistical analyses we carried out on these data. In particular, in Table I for each $W \in \{20{,}50{,}80\}$ we report the results of the statistical comparisons of each configuration against the other five (for a total of $100 \times 6 \times 5 \times 3 = 9{,}000$ comparisons). On the other hand, in Table II we report the results of the statistical comparisons regarding the choice of the value $W \in \{20{,}50{,}80\}$. For the five bloat control and no control at all configurations, we compared each choice of W with the other two (hence $3 \times 2 = 6$ combinations), for a total of $100 \times 6 \times 6 = 3{,}600$ comparisons.

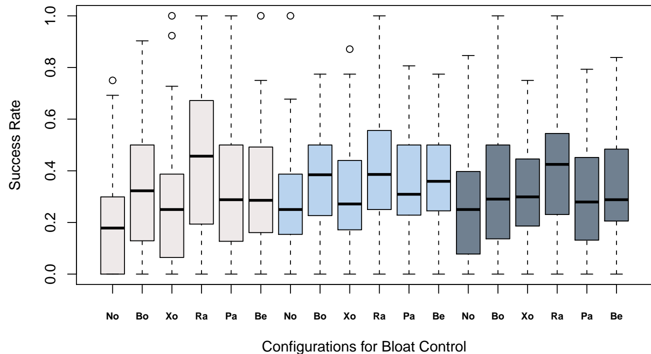


Fig. 4. Success rate for 18 configurations, each applied on all the 100 branches. Left six boxplots are for $W = 20$, $W = 50$ in the centre and $W = 80$ for the six boxplots on the the right of the figure.

TABLE I
COMPARISONS OF BLOAT CONTROL METHODS AGAINST EACH OTHER, WHEN CONSIDERING EACH METHOD IN ISOLATION.

	No	**Bo**	**Xo**	**Ra**	**Pa**	**Be**
Statistically Better	8	49	15	268	51	29
Statistically Equivalent	1347	1393	1392	1224	1402	1402
Statistically Worse	145	58	93	8	47	69

As we can see in those tables and figure, all the controlling bloat techniques have a beneficial effect for obtaining higher success rate. In particular, penalizing longer lengths in rank selection (**Ra**) seems to be the most effective technique regardless of the choice of W.

Regarding the choice of W, we do not see any particular trend in the data. Having short starting sequences ($W = 20$) or long ones ($W = 80$) can have an effect, but that is dependent on the chosen bloat control method (see Table II).

E. Investigations on All Bloat Control Techniques

There can be subtle interactions within the different bloat control methods when more than one is used as the same time. To study these interactions, we carried out the same type of experiments on the remaining $96 - 18$ configurations, for a total of $100 \times 96 \times 25 = 240{,}000$ runs on the algorithm. This is a very large set of experiments that took several days to complete even when run on a cluster of computers.

TABLE II
FOR EACH BLOAT CONTROL METHOD IN ISOLATION, THIS TABLE REPORTS THE NUMBER OF TIMES A PARTICULAR CHOICE OF W PROVIDES BETTER PERFORMANCE THAN THE OTHER TWO.

Bloat Control	$W = 20$	$W = 50$	$W = 80$
No	8	13	24
Bo	20	12	14
Xo	11	17	15
Ra	32	10	18
Pa	18	15	14
Be	16	17	14

To analyze and visualize the results of this large set of data, we followed the following procedure: For each branch, we compared the effectiveness of each configuration against all the other configurations, one at a time (so, 96×95 comparisons, whose calculation can be reduced by half due to the symmetric property of the comparisons). Initially, we assign a score of 0 to each configuration. For each comparison in which a configuration is statistically better, we increase its score by one, and we reduce it by one in case it is statistically worse. Therefore, in the end each configuration has a score between -95 and 95. The higher the score, the better the configuration is.

After this first phase, we rank these scores, such that the highest score has the best rank, where better ranks have lower values. In case of ties, we average the ranks. For example, if we have five configurations with scores $\{10,0,0,20,-30\}$, then their ranks will be $\{2,3.5,3.5,1,5\}$. We repeat this procedure for all the 100 branches, and we calculate the average of these ranks for each configuration, for a total of $100 \times 96 \times 95/2 = 456{,}000$ statistical comparisons.

This a very large number of comparisons, which can lead to a high probability of Type I error [25] if we consider the hypothesis that *all* tests are significant at the same time. We do not use corrections such as the Bonferroni one, for reasons that are discussed in detail and at length in [25]. The configurations with lower average ranks can be considered better then the others. Table III shows the performance of all the 96 configurations, ordered by their ranks.

The results in Table III confirm some of our hypotheses, but also point out some unexpected behaviors. The worst configuration is when no bloat control is activated, and the search starts from small lengths (see last row). This is a particular configuration, with an average success rate 0.190 that is much lower than the 0.464 of the top configuration.

On one hand, regarding the bloat control techniques, the one that has most effect is **Ra**. Activating **Ra** always produces better results, whatever the setting of the other parameters.

On the other hand, it came as a surprise that RPX is actually giving bad results (i.e., **Xo** does not appear in the top rankings). It seems that an unbalanced length crossover such as TPX, which can produce very long as well as very short test cases, is actually beneficial. We can provide a *conjecture* to give a plausible explanation to such an unexpected behavior: In some cases, longer test sequences can have more chances to achieve higher coverage [17], so sampling longer test sequences is beneficial. However, when during the search a fitness plateau is reached, longer test sequences would not have better fitness. Smaller offspring generated with TPX will likely have the same fitness (plateau), but if **Ra** is activated they will be preferred. The search will hence have a sudden drift toward smaller test sequences. Smaller test sequences are quicker to evaluate, and so more generations would be possible. More generations would lead to a more focused search of the test data in input to those sequences, which might help to find the right input data to escape from the plateau. Although this is a plausible explanation, more research will

be required to verify whether that is actual the case, and no other subtle dynamics are involved.

Regarding the other three bloat control methods, they seem beneficial, but they are not as important as **Ra**. When we look at the bottom of the table, it seems that **Pa** is better than **Bo** and **Be**. But at the top, there is not much difference.

The role of W is rather particular: In the best seven configurations, it is set to $W = 50$. But then, for other configurations it does not seem that the choice of W has any particular effect (i.e., we do not see any particular pattern in the data). At the moment, we do not know whether there is any specific reason for why that is the case.

Regarding our initially posed research questions, this leads us to the following conclusions:

RQ1: How does the length influence the search result?

The results of our experiments show that the choice of the maximum starting length W is not particularly important.

Our results have also clearly shown that all bloat control techniques have a large effect of the achieved coverage.

RQ2: How do bloat control methods affect coverage?

Applying bloat control techniques increases coverage significantly.

Among the bloat control methods, **Ra** has a strong beneficial effect, whereas **Xo** decreases the performance. The other three methods are useful, but not as important as **Ra**. Because using only **Ra** alone does not give good enough results (see Table III), based on our results we can suggest the practitioners to use all bloat control methods but **Xo** at the same time.

RQ3: Which bloat control techniques are most effective?

Rank selection with length has the best effect, and should be used together with all other techniques but RPX.

VI. Threats to Validity

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. But it is well known that testing alone cannot prove the absence of defects. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we ran each experiment 25 times, and we followed rigorous statistical procedures to evaluate their results.

The selection of the set of 100 difficult branches was based on only one run for practical reasons (total number of branches was more than 15,000). Therefore, it might be possible that some of them would not be difficult on average, and we could have missed some other difficult branches. However, once the set was selected, all experiments on that set were valid and independent from the chosen selection mechanism.

We used both open source projects and industrial software as case studies, for a total of nearly 1,000 classes. We

TABLE III

Performance of the the 96 configurations, ordered from top (best performance) to bottom (worst performance). Symbols are used to indicate whether a particular bloat control method is activated.

Bo	Xo	Ra	Pa	Be		W		Av. Rank	Av. Success Rate
					20	50	80		
△		⊕	▽	⊞		$\mathcal{W}$		31.475	0.464
△		⊕	▽			$\mathcal{W}$		31.840	0.456
△		⊕		⊞		$\mathcal{W}$		32.595	0.482
		⊕	▽	⊞		$\mathcal{W}$		32.670	0.456
		⊕	▽			$\mathcal{W}$		34.725	0.447
△		⊕				$\mathcal{W}$		35.415	0.448
		⊕		⊞		$\mathcal{W}$		36.070	0.442
△		⊕		⊞	$\mathcal{W}$			37.335	0.423
△	⊠	⊕	▽	⊞		$\mathcal{W}$		37.430	0.430
△		⊕		⊞			$\mathcal{W}$	37.605	0.459
	⊠	⊕		⊞	$\mathcal{W}$			37.615	0.418
△	⊠	⊕		⊞		$\mathcal{W}$		38.080	0.422
	⊠	⊕	▽	⊞		$\mathcal{W}$		39.325	0.419
	⊠	⊕		⊞		$\mathcal{W}$		39.455	0.423
	⊠	⊕	▽			$\mathcal{W}$		39.580	0.413
△		⊕			$\mathcal{W}$			39.790	0.431
		⊕		⊞	$\mathcal{W}$			39.815	0.431
	⊠	⊕			$\mathcal{W}$			40.050	0.414
△		⊕	▽		$\mathcal{W}$			40.140	0.420
△	⊠	⊕	▽			$\mathcal{W}$		40.330	0.425
△		⊕	▽	⊞	$\mathcal{W}$			40.670	0.413
△		⊕	▽	⊞			$\mathcal{W}$	40.700	0.432
△	⊠	⊕		⊞	$\mathcal{W}$			40.835	0.405
		⊕		⊞			$\mathcal{W}$	40.940	0.438
△		⊕	▽				$\mathcal{W}$	41.200	0.455
△	⊠	⊕			$\mathcal{W}$			41.350	0.410
		⊕	▽	⊞			$\mathcal{W}$	41.695	0.423
		⊕	▽	⊞	$\mathcal{W}$			41.890	0.405
		⊕	▽		$\mathcal{W}$			41.925	0.413
	⊠	⊕	▽		$\mathcal{W}$			42.150	0.399
	⊠	⊕	▽	⊞			$\mathcal{W}$	42.195	0.401
	⊠	⊕	▽	⊞	$\mathcal{W}$			42.470	0.388
△	⊠	⊕	▽		$\mathcal{W}$			42.500	0.395
	⊠	⊕		⊞			$\mathcal{W}$	42.800	0.422
		⊕			$\mathcal{W}$			43.075	0.407
	⊠	⊕				$\mathcal{W}$		43.095	0.421
△	⊠	⊕				$\mathcal{W}$		43.255	0.420
△	⊠	⊕	▽	⊞	$\mathcal{W}$			43.635	0.377
		⊕				$\mathcal{W}$		45.160	0.398
	⊠	⊕	▽				$\mathcal{W}$	45.205	0.393
		⊕	▽				$\mathcal{W}$	45.285	0.412
△	⊠	⊕	▽				$\mathcal{W}$	45.450	0.392
△		⊕					$\mathcal{W}$	45.850	0.418
		⊕					$\mathcal{W}$	46.460	0.401
△	⊠	⊕					$\mathcal{W}$	46.625	0.388
△	⊠	⊕		⊞			$\mathcal{W}$	46.700	0.409
△	⊠	⊕	▽	⊞			$\mathcal{W}$	47.760	0.379
	⊠	⊕					$\mathcal{W}$	47.850	0.384
△			▽	⊞		$\mathcal{W}$		48.985	0.342
			▽			$\mathcal{W}$		49.585	0.329
			▽	⊞		$\mathcal{W}$		49.705	0.334
△			▽	⊞	$\mathcal{W}$			49.995	0.369
△	⊠		▽	⊞		$\mathcal{W}$		50.290	0.313
△			▽		$\mathcal{W}$			50.740	0.356
△	⊠		▽			$\mathcal{W}$		51.295	0.313
△			▽			$\mathcal{W}$		51.350	0.340
△				⊞		$\mathcal{W}$		51.570	0.327
△			▽	⊞			$\mathcal{W}$	52.215	0.326
△				⊞	$\mathcal{W}$			52.800	0.330
			▽	⊞			$\mathcal{W}$	53.260	0.330
	⊠		▽	⊞		$\mathcal{W}$		53.610	0.309
△			▽				$\mathcal{W}$	53.845	0.321
	⊠		▽	⊞	$\mathcal{W}$			54.040	0.310
	⊠		▽				$\mathcal{W}$	54.475	0.312
			▽	⊞	$\mathcal{W}$			54.835	0.296
			▽		$\mathcal{W}$			55.080	0.306
				⊞		$\mathcal{W}$		55.290	0.317
	⊠		▽			$\mathcal{W}$		55.390	0.313
	⊠		▽	⊞			$\mathcal{W}$	55.605	0.304
△					$\mathcal{W}$			55.635	0.305
			▽				$\mathcal{W}$	55.695	0.324
△	⊠		▽		$\mathcal{W}$			56.065	0.310
△						$\mathcal{W}$		56.160	0.309
	⊠			⊞		$\mathcal{W}$		56.200	0.304
△	⊠		▽	⊞			$\mathcal{W}$	56.255	0.301
	⊠		▽		$\mathcal{W}$			56.295	0.312
△	⊠		▽	⊞	$\mathcal{W}$			56.655	0.312
△	⊠		▽				$\mathcal{W}$	56.835	0.291
△	⊠					$\mathcal{W}$		57.095	0.279
△	⊠			⊞		$\mathcal{W}$		57.135	0.291
△				⊞			$\mathcal{W}$	57.180	0.319
				⊞			$\mathcal{W}$	57.390	0.306
							$\mathcal{W}$	58.955	0.285
△	⊠			⊞			$\mathcal{W}$	59.085	0.297
				⊞	$\mathcal{W}$			59.190	0.297
△	⊠			⊞	$\mathcal{W}$			59.270	0.285
	⊠					$\mathcal{W}$		59.595	0.279
△							$\mathcal{W}$	59.995	0.300
	⊠			⊞	$\mathcal{W}$			60.145	0.281
	⊠						$\mathcal{W}$	60.150	0.289
△	⊠				$\mathcal{W}$			60.675	0.278
	⊠			⊞			$\mathcal{W}$	60.705	0.289
△	⊠						$\mathcal{W}$	60.975	0.292
						$\mathcal{W}$		61.655	0.267
	⊠				$\mathcal{W}$			65.220	0.238
					$\mathcal{W}$			71.765	0.190

selected different types of applications, such as for example implementations of data structures, complex manipulations of string data and numerical applications. Nevertheless, there is still the threat to *external validity* regarding the generalization to other types of software, which is common for any empirical analysis. Furthermore, due to the large amount of experiments, only 100 branches were used as case study.

VII. Conclusions

Evolutionary search with variable size representation is susceptible to *bloat*—that is, a disproportional growth of the length of individuals that quickly uses up all resources and so seriously harming the search. Unfortunately, this also means it applies to search-based testing for object-oriented software, although this has not been sufficiently treated in the literature so far. In this paper, we performed a set of experiments, using a genetic algorithm, on the properties of test sequence *length* and how to counter the effects of length *bloat*.

Interestingly, our results showed that the choice of starting length for the search is secondary to the choice of bloat control techniques. Not only is there the danger of running into problems such as using up all memory and increasing execution times, our experiments showed that the success rate for the same amount of resources is *significantly* higher when applying bloat control techniques.

Our experiments clearly points to which bloat control techniques to use and which ones should not be used in practical contexts. As long as bloat is properly taken under control, the choice of test sequence lengths at the beginning of search is not particularly important, because the search algorithms can learn which are the best lengths toward direct the search effort. To support our claims, we carried out a very large empirical study on several types of applications (e.g., data structures, string processing, numerical applications), using both open source and industrial software. A rigorous statistical method has been employed to verify with high confidence that our results are scientifically valid.

As future work, we plan to apply our experiments to a larger set of case studies, and include further coverage criteria in addition to branch coverage. In addition, we need to consider the effects of other parameters such as the population size in genetic algorithms.

Acknowledgments. Gordon Fraser is funded by the Cluster of Excellence on Multimodal Computing and Interaction at Saarland University, Germany. Andrea Arcuri is funded by the Norwegian Research Council.

References

[1] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[2] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2009.

[3] S. Silva and E. Costa, "Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories," *Genetic Programming and Evolvable Machines*, vol. 10, no. 2, pp. 141–179, 2009.

[4] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," CREST Centre, King's College London, London, UK, Technical Report TR-09-06, September 2009.

[5] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 438–444, 1984.

[6] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, pp. 385–394, 1976.

[7] P. Tonella, "Evolutionary testing of classes," in *ISSTA'04: Proceedings of the ACM International Symposium on Software Testing and Analysis*. ACM, 2004, pp. 119–128.

[8] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *ISSTA'10: Proceedings of the ACM International Symposium on Software Testing and Analysis*. ACM, 2010, pp. 147–158.

[9] S. Wappler and F. Lammermann, "Using evolutionary algorithms for the unit testing of object-oriented software," in *GECCO'05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*. ACM, 2005, pp. 1053–1060.

[10] J. C. B. Ribeiro, "Search-based test case generation for object-oriented Java software using strongly-typed genetic programming," in *GECCO'08: Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*. ACM, 2008, pp. 1819–1822.

[11] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," in *ASE'08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 297–306.

[12] A. Arcuri and X. Yao, "Search based software testing of object-oriented containers," *Information Sciences*, vol. 178, no. 15, pp. 3075–3095, 2008.

[13] L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: an evolutionary test approach for Java," in *ICST'10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2010, pp. 185–194.

[14] J. H. Andrews, T. Menzies, and F. C. Li, "Genetic algorithms for randomized unit testing," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2010.

[15] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley and Sons, 2001.

[16] M. Harman and P. McMinn, "A theoretical and empirical study of search based testing: Local, global and hybrid search." *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2010.

[17] A. Arcuri, "Longer is better: On the role of test sequence length in software testing," in *ICST'10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2010, pp. 469–478.

[18] J. H. Andrews, A. Groce, M. Weston, and R. G. Xu, "Random test run length and effectiveness," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008, pp. 19–28.

[19] G. Fraser and A. Gargantini, "Experiments on the test case length in specification based test case generation," in *International Workshop on Automation in Software Test (AST)*, 2009.

[20] D. Whitley, "The GENITOR algorithm and selective pressure: Why rank-based allocation of reproductive trials is best," in *ICGA'89: Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989, pp. 116–121.

[21] A. Arcuri, "It does matter how you normalise the branch distance in search based software testing," in *ICST'10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2010, pp. 205–214.

[22] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators: Research articles," *Software Testing, Verification, and Reliability*, vol. 16, no. 3, pp. 175–203, 2006.

[23] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness," Simula Research Laboratory, Tech. Rep. 2010-09, 2010.

[24] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," in *ICTSS'10: Proceedings of the IFIP International Conference on Testing Software and Systems*. Springer, 2010.

[25] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," Simula Research Laboratory, Tech. Rep. 2010-10, 2010.