

Search Based Testing of Containers for Object-Oriented Software

Andrea Arcuri and Xin Yao
The Centre of Excellence for Research
in Computational Intelligence and Applications (CERCIA)
The School of Computer Science
The University of Birmingham
Edgbaston, Birmingham B15 2TT, UK
{A.Arcuri,X.Yao}@cs.bham.ac.uk

Abstract

Automatic software testing tools are still far from ideal for real world Object Oriented (OO) Software. The use of search algorithms that exploit the branch distances for this problem has been investigated recently. However, no specific work on Container classes has been reported, yet containers are used in almost every type of software. Hence, their reliability is of utmost importance. Testing complex data structures (containers) is very challenging since testing software with simple states is already hard. This paper will first describe how input data can be automatically generated for testing Java Containers. Input space reductions and a novel testability transformation are presented to aid the search algorithms. Different search algorithms are then considered and studied in order to understand when and why a search algorithm is effective for a testing problem. Finally, some open research questions are given.

1 Introduction

Modern software projects have become increasingly large and complex. It is not difficult to find programs of millions of lines of code, with hundreds of different engineers having worked on them. In spite of years of research effort in software engineering, software is still far from being perfect and still suffers badly from various bugs.

Finding and fixing bugs become a very expensive and time consuming issue for the entire software engineering industry. It is impossible to guarantee a bug-free software unless it is trivially small. Various techniques have been proposed to minimise the possibility of bugs, e.g., trying to identify them as early as possible in the software development cycle. One way to try to deal with this issue is the use of *Unit Tests* [14]. It consists of writing pieces of code that test every class and method of the project. For example, a method that sums two integers can be called with two particular values (1 and 2 for example), and then the result will be checked against the expected value (3). If they do not match, we can be sure that there is something wrong with the code. The above is a trivial example, but we can understand the real power of unit tests when we think about

more complex functions, that can be changed in different periods (often in different years) by different programmers. Besides, if a new feature or bug fix is added, with a unit test we can discover if it breaks or not the former functionality. Writing unit tests requires time and resources, and usually it is a tedious job. Thus, a way to automatically generate unit tests is needed. Different approaches have been used to solve this problem, but a system that can generate a complete set of unit tests for a generic OO program has not been developed yet. More research on that subject is required.

This paper presents a framework for automatically generating unit tests for *Container* classes. The test criterion is *White Box Testing*. There is no previous work on this problem that use search algorithms that exploit the branch distances. Therefore, this paper analyses different search algorithms. We use a search space reduction that exploits the characteristics of the containers. Without this reduction, the use of search algorithms would have required too much computational time. This paper also presents a novel testability transformation [11] to aid the search algorithms. Although the programming language used is Java, the techniques described in this paper can be easily applied to other object oriented languages.

In section 2 a particular class of software (containers) with its properties is presented. Section 3 describes how to apply four different search algorithms to automatically generate unit tests for containers. Experiments carried out on the proposed algorithms follow in section 4. A short review of the related literature is given in section 5. The conclusions of this work can be found in section 6.

2 Testing of Java Containers

In OO programs, containers hold an important role because they are widely use in almost any type of software. The techniques described in this paper are applied to the Java language, but they can easily be extended to other OO languages (*C++* for example). Not only do we need to test novel types of containers and their new optimisations, but also the current libraries need to be tested [21].

There are different types of containers, like Arrays, Vectors, Lists, Trees, etc. We usually expect from a container methods like `insert`, `remove` and `find`. Although the interfaces of these methods can be the same, how they are implemented and their computational cost can be very different. Besides, the behaviour of such methods depends on the elements already stored inside the container. The presence of an internal state is a problem for software testing [24, 26, 22], and this is particularly true for the containers.

2.1 Properties of the problem

A solution to the problem is a sequence S_i of function calls (FCs) on an instance of the container under test (CuT).

A Function Call (FC) can be seen as a triple:

$\langle \textit{object_reference}, \textit{function_name}, \textit{input_list} \rangle$

It is straightforward to map a FC in a command statement. E.g.:

```
ref.function(input[0], ..., input[n]);
```

I.e., given an *object_reference* called `ref`, the function with name *function_name* is called on it with the input in *input_list*. There will be only one S_i for the CuT, and not one for each of its branches (as usually happens in literature). Each FC is embedded in a different `try/catch` block. Hence, also the paths that throw exceptions can be covered.

Given a coverage criterion, we are looking for the shortest sequence that gives the best coverage. For simplicity, we will consider only “branch coverage”. However, the discussion can be easily extended to other coverage criteria.

Let S be the set of all possible solutions to the given problem. The function $cov(S_i) : S \rightarrow N$ will give the coverage value of the sequence S_i . That function will be upper bounded by the function $max(container) : Containers \rightarrow N$, so $cov(S_i) \leq max(CuT) \forall S_i \in S$. The function max gives in output the number of branches in a container. It is important to remember that the best coverage (global optimum) can be lower than max , due to the fact that some paths in the execution can be infeasible. The length of sequence is given by $len(S_i) : S \rightarrow N$. We prefer solution S_i to S_j iff the next predicate is true:

$$\begin{cases} cov(S_i) > cov(S_j) \\ cov(S_i) = cov(S_j) \wedge len(S_i) < len(S_j) \end{cases} \quad \text{or} \quad (1)$$

The problem can be viewed as a Multi-Objective task. A sequence with a higher coverage will always be preferred regardless of the length. Only if the coverage is the same, we will look at the length of the sequences. Thus, the coverage can be seen as a hard constraint, and the length as a soft constraint. Although there are two different objectives, coverage and length, it is not a good idea to use Pareto dominance in the search. This is because the length is always less important than the coverage. Thus, the two objectives should be properly combined together in a single fitness function, as for example:

$$f(S_i) = cov(S_i) + \frac{1}{1 + len(S_i)}. \quad (2)$$

Containers have some specific properties. Some of them depend on the the actual implementation of the container, thus they cannot be analytically solved without considering the source code. However, our empirical tests show that these properties are true for any examined containers. Let k be the position in S_i of the last FC that improve the coverage of that sequence. The properties are:

- any operations (like insertion, removal and modification of one FC) done on S_i after the position k cannot decrease the coverage of the sequence. This is always true for any container.
- given a random insertion of an FC in S_i , the likelihood that the coverage decreases is low. It will be always zero if the insertion is after k , or if the FC does not change the state of the container.
- following from the previous property, given a random removal of a FC in S_i , the likelihood of the coverage increases is low.
- The behaviour of a FC depends only on the state of the tested container when the FC is executed. Therefore, a FC cannot alter the behaviour of the already executed FCs.
- let S_r be a sequence, and S_t be generated by S_r adding any FC to the tail of S_r . Due to the previous property, we have $cov(S_t) \geq cov(S_r)$.

For the same reason, we have $cov(S_r, i) \geq cov(S_r, j)$, where $j \leq i \leq len(S_r)$ and $cov(S_r, i)$ gives the coverage of S_r when only the first i FCs are executed. Given two random sequences generated with the same distribution, therefore, it is easy to understand that we should expect a better coverage from the longest one.

- calculating the coverage for S_i requires the call of $len(S_i)$ methods of the CuT. This can be very expensive from a computational point of view. Thus, the performance in terms of a search algorithm depends heavily on the length of the sequences that it evaluates.

2.2 Search Space Reduction

The solution space of the test sequences for a container is very huge. We have M different methods to test, so there are M^L possible sequences of length L . We do not know a priori what is the best length. Although we can put an upper bound to the max length that we want to consider, it is still an extremely large search space. Besides, each FC can take some parameters as input, and that increases the search space even further.

The parameters in input for the FCs make the testing very difficult. They can be references to instances of objects in the heap. Not only the set of all possible objects is infinite, but a particular instance may need to be put in a particular state by a sequence of FCs on it (they are different from the ones of the container). In the same way, these latter FCs can require objects in input which need sequences of their proper FCs on them.

Fortunately, for testing of Java Containers, we can limit the space of the possible inputs for the FCs. In fact, usually the methods of a container need as input only the following types:

1. *indices* that refer to a position in the container. A typical method that uses them is `get(int i)`. The indices are always of `int` type. Due to the nature of the containers, we just need to consider values inside the range of the number of elements stored in the CuT. Besides, we also need some few indexes outside this range. If we consider that a single FC can add no more than one element to the container (it is generally true), the search space for the indices can be bound by the length of S_i .
2. *elements* are what are stored in a container. Usually they are references to objects. In the source code, the branch statements that depend on the states of elements are only of three type: the *Natural Order* between the elements, the equality of an element to `null` and the belonging of it to a particular Java class. This latter type will be studied only in future work. The natural order between the elements is defined by the methods `equals` and `compareTo`. Given n elements in the CuT, we require that all the orders of these n elements are possible. We can easily do it by defining a set Z , with $|Z| = n$, such that all elements in Z are different between them according to the natural order (i.e., if we call `equals` to any two different elements in Z we should get always false as a result). A search algorithm can be limited to use only Z for the elements in input for the FCs without losing the possibility of reaching the global optimum. Anyway, we should add to Z the value `null`.

The number n of elements in the CuT due to S_i is upper bounded by $len(S_i)$ (we are not considering the possibility that a FC can add more than one element to the CuT). Because the natural order does

not depend on the container, we can create Z with $|Z| \leq \text{len}(S_i) + 1$ regardless of the CuT. We can as example use `Integer` objects with value between 0 and $|Z|$. It is important to outline that Z is automatically generated : the user does not have to give any information to the system. Besides, the size of Z dynamically changes with the change of the length of the sequence under test.

3. *keys* are used in containers such as *Hash Tables*. The considerations about *elements* can be also applied to the *keys*. The difference is that the method `hashCode` can be called on them. Because how the hash code is used inside the source code of the CuT cannot be known a priori, we cannot define for the keys a finite set such Z that guarantees us that the global optimum can be reached. In our experiments, we used the same Z for both *elements* and *keys* with good results. However, for the *keys*, there are no guaranties that Z is enough.
4. some FCs may need in input a reference to one other container. Such type of FCs are not considered at the moment, and they are excluded from the testing. Future work will be done to cover also them. However, in our test cluster, only 11 methods on a total of 105 need this type of input.

All the search algorithms that will be analysed in this paper will use the space reductions described above, unless otherwise stated.

2.3 Branch Distance

If the only information that a search algorithm can have is about whether a branch is covered or not, the algorithm will have no guidance on how to cover it. In other words, the search space will have big plateaus. In such a case, we can say that the search is “blind”. Anyway, if the predicate of that branch statement is not complex, the likelihood of cover both its branches by chance is not so low. It is for that reason that a random search can achieve a quite good coverage. When the predicate is complicated, a blind search is likely to fail. Let’s say that we want to cover a *then* branch of an *if* statement (the *else* branch can be studied in the same way). One way to address this problem is to use the *Branch Distance* (BD) of the branch [17]. That is a real value that tells us how far the predicate is to be evaluated as true. If we include BD to the coverage value of the sequence, the search space will become smoother. There can be different ways to define BD. A widely used way can be found in [31].

The BD has an important role in Software Testing, and a lot of research has been done on how to improve its effectiveness [3]. Common issues are: *flags* in the source code [10, 2], especially if they are assigned in loops [1], dependences of the predicate to previous statements [9, 25] and nested branch statements [23]. Anyway, those works are concerning the coverage of only one particular branch at time. On the other hand, the test sequences for Java Containers analysed in this paper try to cover all the branches at the same time. According to [37], it should be expected a better performance if each predicate is target/tested separately. I.e., there will be a different test driver for each predicate in the CuT. However, that work does not consider the necessity of a sequence of function calls to put the program in a helpful state for testing. To our best knowledge, the use of a single sequence of FCs to cover all the branches at the same time with

the aid of the branch distances has never been analysed before. Therefore, a comparison of the two different techniques is required.

In detail, we need to use:

$$b(j) = \begin{cases} 0 & \text{if the branch } j \text{ is covered ,} \\ k & \text{if the branch } j \text{ is not covered and its} \\ & \text{opposite branch is covered at least twice ,} \\ 1 & \text{otherwise ,} \end{cases} \quad (3)$$

$$0 < k < 1 , \quad (4)$$

$$B(S_i) = \sum_{j=1}^{\max(\text{CuT})} \frac{b(j)}{\max(\text{CuT})} . \quad (5)$$

Where k is the lowest normalised BD for the predicate j during the execution of S_i . The function $b(j)$ defined in (3) describes how close the sequence is to cover that not covered branch j . If its branch statement is never reached, it should be $b(j) = 1$ because we cannot compute the BD for its predicate. Besides, it should also be equal to 1 if the branch statement is reached only once. Otherwise, if j will be covered due to the use of $b(j)$ during the search, necessarily the opposite of j will not be covered any more (we need to reach the branch statement at least twice if we want to cover both of its branches).

We can integrate the normalised BDs of all the branches (5) with the coverage of the sequence S_i in the following way:

$$cb(S_i) = cov(S_i) + (1 - B(S_i)) . \quad (6)$$

It is important to notice that such a function (6) guarantees that

$$cb(S_i) \geq cb(S_j) \Rightarrow cov(S_i) \geq cov(S_j) . \quad (7)$$

Finally, to decide if S_i is better than S_j , we can replace (1) with:

$$\begin{cases} cb(S_i) > cb(S_j) \\ cb(S_i) = cb(S_j) \wedge len(S_i) < len(S_j) . \end{cases} \quad \text{or} \quad (8)$$

In such a way, the search landscape gets smoother. Although we can use (8) to aid the search, we still need to use (1) to decide which is the best S_i that should be given as the result. In fact, for the final result, we are only interested in the achieved coverage and length. How close the final sequence is to get a better coverage is not important.

2.4 Testability Transformations

When BDs are used, a common way to improve the performance of a search algorithm is the use of *Testability Transformations* [11]. They are source to source transformations applied to the code of the CuT. Their only requirement is that the test data generated to the transformed program are adequate to the original one. Usually, these transformations are used to handle the *flags* problem. Here we propose a novel testability transformation that is specific for the Java Containers.

Some branch statements in a Java Container source code rely on the natural order of its stored elements. The method `equals` gives no information on how much two elements are different, because it returns only a boolean value (this is like the flag problem). Therefore, we propose to transform all calls to the `equals` method:

```
obj.equals(other)
```

with:

```
((Comparable)obj).compareTo(other) == 0)
```

We should notice that this transformation is useful only if the stored objects implement the interface `Comparable`. If we use the set Z defined in sec 2.2, we can guarantee it. Although the method `compareTo` says if two objects are different, it does not say how much they are different. Therefore, we need that the objects in Z extend this semantic by subclassing this method. Because Z can be implemented regardless of the container that will be tested, besides the fact that `compareTo` gives an integer as its output, we can easily do this extension of the semantic. For example, `compareTo` can give as output the difference between the natural order values of the two compared objects. Hence, the search landscape gets smoother, and this fact helps the search algorithms.

It is arguable why we use `compareTo` instead of defining a new method. This choice was made because in such a way we need no testability transformations on the `compareTo` calls. I.e., we need to transform only the calls to `equals` and not the ones to `compareTo`. Otherwise, the `compareTo` method with its original semantic generates plateaus in the search space. However, we cannot know (or it is too difficult to be exactly determined) if all calls to `equals` belong only to the stored elements. Thus, to be sure of not having any exception raised during the search, we can use this other transformation instead:

```
((obj instanceof Comparable) ?  
 ((Comparable)obj).compareTo(other) :  
 (obj.equals(other) ? 0 : 1))  
 == 0 )
```

However, this latter transformation introduces two new branches in the program for every call to `equals`. This is a problem, because input test data (in our case a sequence of FCs) for branch coverage for the transformed program is not adequate for the original one. Given a support library `ett` (the name is not important), one possible solution is:

```
(ett.Utils.compare(obj,other)==0)
```

with the function `compare` implemented in this way:

```
public static int compare(Object obj, Object other)  
{  
    if(obj instanceof Comparable)  
        return ((Comparable)obj).compareTo(other);  
    else  
        return obj.equals(other) ? 0 : 1;  
}
```

It is important that the instrumentation tool (described in 2.5) should be informed to ignore this function (i.e., that function should be considered as a side effect free external function of which there is no access to its source code).

This testability transformation can be done automatically, without any help of the user.

2.5 Instrumentation of the Source Code

To guide a search algorithm, the CuT needs to be executed to get the branch distances of the predicates [17]. To analyse the execution flow, the source code of the CuT needs to be *instrumented*. That means that extra statements will be added inside the original program. To trace the branch distances, we need to add a statement (usually a function call) before every predicate. These statements should not alter the behaviour of the CuT, i.e. they should not have any side effect on the original program. They should only compute the branch distance and inform the testing environment of it. If a constituent of a predicate has a side effect (like the ++ operator or a function call that can change the state of the program), testability transformation should be used to remove the side effects [12]. Otherwise, such side effect could be executed more than once, changing the behaviour of the program.

In the environment that has been developed, it was chosen to use the program *srcML*[7] for translating the source code of the CuT in an XML representation. All instrumentations and testability transformations are done on the XML version. Afterwards, *srcML* is used to get the modified Java source code from the XML representation. This new source code is used only for testing purposes. It will be discarded after the testing phase is finished. The human tester should not care about this transformed java source code.

3 Search Algorithms

There is not a search algorithm that outperforms all the other algorithms [38]. Therefore, when a new problem needs to be addressed, different simple algorithms should be compared. That facilitates a deeper understanding of the search problem, so that more appropriate algorithms can be developed as the next step. In this paper four search algorithms are used: Random Search, Hill Climbing, Simulated Annealing and Genetic Algorithms. Although there are different variants of these algorithms, in this paper only some simple forms are investigated.

3.1 Using Random Search

Although Random Search is the easiest among the search algorithms, it may give good coverage. The only problem is that we need to define the length of sequences that will be generated during the search. If we do it in a pure random way, there can be extremely long sequences. Not only the computational time can be too expensive, but also such long sequences will be useless. Therefore, it is better to put an upper bound L to the length. The sequences can still have a random length, but it will be always lower than L . Random search can be implemented in the following way:

1. generate at random sequence S_i , with $len(S_i) < L$.
2. compare S_i with the best sequence seen so far. For comparison, use formula (1). If S_i is better, store it as the new best.
3. if the search is not finished (due to time limit or number of sequences evaluated), go to step 1.
4. return the best sequence seen so far.

Note that the only parameter that needs to be set is the upper bound L . The *branch distance* technique is useless in a random search.

3.2 Using Hill Climbing

Hill Climbing (HC) is a local search algorithm. It needs that a neighbourhood N of the current sequence S_i is defined. The search will move to a new solution $S_j \in N$ only if S_j is better. If there are no better solutions in N , a local optimum has been reached. In such a case, a restart of the algorithm from a new random solution can be done.

We need to define N . Its size has a big impact on the performance of the algorithm [41]. We can think of three types of operations that we can do on S_i for generating N . Others types of operations will be investigated in future work.

Removal of a single FC from S_i . There will be $len(S_i)$ different neighbour sequences due to this operation.

Insertion of a new FC in S_i . There are $len(S_i) + 1$ different positions in which the insertion can be done. For each position, there are M different methods that can be inserted. Due to the too large search space, the input parameters for the FC will be generated at random.

Modification of the parameters of a FC. All FCs in S_i will be considered, except for the FCs with no parameters. If a parameter is an *index* i (see 2.2), we will have two operations that modified i by ± 1 . Otherwise, if the parameter belongs to Z , we will consider two operations which will replace the parameter with the two closest elements in Z according to their natural order.

The branch distance should be used carefully. In fact, if (8) is used, the HC can be driven to increase the length to try to cover a particular difficult branch. If it falls in a local optimum without covering that branch, the result sequence can be unnecessarily too long. Because HC finds a local optimum, it cannot decrease the length of the sequence. We can resolve this problem by continuing the search from that local optimum using (1) instead.

3.3 Using Simulated Annealing

The use of Simulated Annealing (SA) [16] has been investigated. A variant of SA that uses Metropolis procedure has been used. The neighbourhood of the current sequence N is defined in the same way as in the Hill Climbing algorithm.

It is not easy to define the *energy* of a sequence S_i . According to the Metropolis procedure, the energy is used to compute the probability that a worse sequence will be accepted. Such probability is

$$W = \exp\left(-\frac{E(S_{i+1}) - E(S_i)}{T}\right). \quad (9)$$

The problem is that we need to properly combine in a single number two different objectives as the coverage and the length of a given sequence. We need that, if S_i is better than S_j according to (1), than $E(S_i) < E(S_j)$. In formulae:

$$E(S_i) = f(cov(S_i)) + g(len(S_i)), \quad (10)$$

$$f(max(CuT)) = 0, \quad (11)$$

$$cov(S_i) > cov(S_j) \Rightarrow f(cov(S_i)) < f(cov(S_j)) \forall S_i, S_j \in S, \quad (12)$$

$$cov(S_i) > cov(S_j) \Rightarrow E(S_i) < E(S_j) \forall S_i, S_j \in S, \quad (13)$$

$$\text{len}(S_i) > \text{len}(S_j) \Rightarrow g(\text{len}(S_i)) > g(\text{len}(S_j)) \quad \forall S_i, S_j \in S . \quad (14)$$

The function f is used to weigh the coverage, and it can easily be written as:

$$f(S_i) = \max(CuT) - \text{cov}(S_i) . \quad (15)$$

On the other hand, the function g weighs the length. Due to (13) and the fact that the coverage is a positive natural value, we have:

$$0 \leq g(\text{len}(S_i)) < 1 \quad \forall S_i \in S . \quad (16)$$

It should be less than 1, otherwise it is possible that a sequence can have a lower energy comparing to a longer sequence with higher coverage. One way to do it is :

$$g(\text{len}(S_i)) = 1 - \frac{1}{1 + \text{len}(S_i)} . \quad (17)$$

Using (15) and (17), we can rewrite (10) as:

$$E(S_i) = \max(CuT) + 1 - \left(\text{cov}(S_i) + \frac{1}{1 + \text{len}(S_i)} \right) . \quad (18)$$

Although the latter formula defines the energy of a sequence in a proper way, it should not be used in the Simulated Annealing algorithm with the neighbourhood described before. This energy can deceive the search letting it to look at sequences always longer at every step. Consider the case Q in which a potential new sequence K is generated by S_i by adding a FC that does not change the coverage, but that increase the length. Thus, $\text{cov}(K) = \text{cov}(S_i)$ and $\text{len}(k) = \text{len}(S_i) + 1$. The new sequence K will be accepted as S_{i+1} by the probability (9). Therefore,

$$W_{Q,i} = \exp \left(- \frac{1}{L^2 \cdot T} \right) , \quad (19)$$

$$L^2 = \text{len}(S_i)^2 + 3 \cdot \text{len}(S_i) + 2 . \quad (20)$$

If the temperature T decreases slowly (as it should be), it is possible that $W_{Q,i+1} > W_{Q,i}$ because the length of sequence has increased. Although it should be mathematically proved, this behaviour has a high likelihood to tend to increase the length of the sequence. Empirical tests confirm it.

Instead of (17), we can think to use something like $g(\text{len}(S_i)) = \frac{\text{len}(S_i)}{M}$. However, this is not a good idea. In fact, we cannot set any finite M such that (13) is always true. Although we can limit M to the maximum length that the actual machine can support/handle, we will have in such a case that the weight of the length has little impact on the energy for any reasonable value of the length. I.e., $g(\text{len}(S_i))$ will be very close to zero for any S_i encountered during the search.

For all these reasons, the Simulated Annealing algorithm used is slightly different from the common version. Regardless of any energy, a new sequence K will be always accepted as S_{i+1} if (1) is true. Otherwise, the Metropolis procedure will be used. The energy function will be:

$$E(S_i) = \max(CuT) - \text{cov}(S_i) + \alpha \cdot \text{len}(S_i) . \quad (21)$$

The constant α has a very important role on the performance of the SA. Before studying what is its best value, we need some assumptions:

- the likelihood that the sequence K will be generated using a *removal* operation on S_i is the same as having an *insertion*. I.e., $P(rem) = P(ins)$.
- the operations used to generate K can increase or decrease the length only by one. I.e., $len(K) - len(S_i) \in \{-1, 0, 1\}$.
- all assumptions in section 2.1 hold.

There is a big problem that, due to (21), it is possible that there can be no change in the energy (i.e. $W = 1$) even if the new state is worse according to (1). In such cases, some particular worse sequences will always be accepted regardless of the temperature T . We will firstly analyse the values of α for which that thing does not happen. Afterwards, we will study how the SA behaves when α is not a chosen properly. There are different situations:

1. $len(S_{i+1}) - len(S_i) = 0$. According to (1), we have that $cov(S_{i+1}) \leq cov(S_i)$, otherwise the new sequence would have already been accepted. Therefore, $W \geq 1$ iff $cov(S_{i+1}) = cov(S_i)$. That means that if there are no changes in both the coverage and length, the new sequence S_{i+1} will always be accepted. This is not a problem for SA, but for other algorithms as Hill Climbing this rule can generate an infinite loop in the search. In this scenario, the value of α has no influence.
2. if $cov(S_{i+1}) = cov(S_i)$, we still need to discuss when $len(S_{i+1}) - len(S_i) = 1$. Note that, due to (1), it cannot be minus one. In that case, we have $W = \exp\left(-\frac{\alpha}{T}\right)$. So we need $\alpha > 0$.
3. $cov(S_{i+1}) < cov(S_i)$ and $len(S_{i+1}) - len(S_i) = 1$. In such a case, we can set $\alpha \geq 0$ to guarantee that $W < 1$.
4. the worst case is when $cov(S_{i+1}) < cov(S_i)$ and $len(S_{i+1}) - len(S_i) = -1$. In fact, one objective (the coverage) gets worse, but at the same time the other objective (the length) gets better. For having $W < 1$, we need that $E(S_{i+1}) - E(S_i) = cov(S_i) - cov(S_{i+1}) - \alpha > 0$. Therefore, given

$$M = \min(cov(S_i) - cov(S_{i+1})), \quad (22)$$

$$\forall S_i \in S \quad \forall S_{i+1} \in \{S_j | S_j \in N_{S_i}, cov(S_j) < cov(S_i)\},$$

we should have:

$$\alpha < M. \quad (23)$$

In our case, we have $M = 1$ because the coverage is always a natural value. Thus, $\alpha < 1$.

The range of value for α such that all the previous conditions are true at the same time is :

$$0 < \alpha < M, \quad (24)$$

with $M = 1$. However, it is important to notice that, for the energy defined in (21), we cannot use the *branch distance*. Otherwise, the only lower bound for M will be zero. In such a case, we will have $0 < \alpha < 0$ that has no solution. However, there are no problems to use (8) instead of (1) for deciding if a new sequence is better or not.

If α is chosen not in the range defined in (24), the SA algorithm will be deceived.

$\alpha \geq 1$, any neighbour S_j of S_i with worse coverage, shorter length and $cov(S_j) + \alpha \geq cov(S_i)$, will always be accepted as S_{i+1} regardless of the temperature T . Because it is common that any global optimum for a generic container has at least one FC that contributes to the coverage only by one, the SA with $\alpha \geq 1$ cannot be guaranteed to converge. Besides, from empirical tests, the performances of the SA are so poor that even a Random Search performs better than it.

$\alpha \leq 0$, any neighbour S_j that does not reduce the coverage will always be accepted, even if the length increases. Although it does not seem a problem because a S_j with same coverage but shorter length is always accepted due to (1), the SA will tend to move its search to sequences always longer and longer. That can be explained if we consider the probabilities that S_{i+1} has been generated by S_i with an insertion or by removing a FC. Given an operation (op), the probability that applying it to S_i the new sequence will be accepted (acc) is:

$$P(acc|op)_i = \begin{cases} 1 & \text{if } cov(S_{i+1}) \geq cov(S_i) \text{ ,} \\ W_i & \text{otherwise .} \end{cases} \quad (25)$$

Due to the assumption in section (2.1), the probability of an accepted insertion (ins) is:

$$P(acc|ins) \approx 1 . \quad (26)$$

On the other hand, the probability that a removal (rem) is accepted is:

$$P(acc|rem) \approx P(cov(S_{i+1}) = cov(S_i)|rem) + W_i \cdot P(cov(S_{i+1}) < cov(S_i)|rem) . \quad (27)$$

Let R_i be the number of redundant FCs in S_i , i.e. if we remove any of this FCs the coverage does not change. We will have:

$$P(cov(S_{i+1}) = cov(S_i)|rem) = \frac{R_i}{len(S_i)} , \quad (28)$$

$$P(cov(S_{i+1}) < cov(S_i)|rem) = \frac{len(S_i) - R_i}{len(S_i)} . \quad (29)$$

Therefore, using (28) and (29), we can write (27) as:

$$P(acc|rem) \approx W_i + (1 - W_i) \cdot \frac{R_i}{len(S_i)} . \quad (30)$$

Unless we want to change how the neighbourhood is defined, i.e. $P(rem) = P(ins)$, it is more likely that S_{i+1} has been generated from an insertion operation because $R_i < len(S_i)$. I.e. ,

$$P(ins|acc) > P(rem|acc) . \quad (31)$$

This is particularly true when S_i is close to a local optimum for the length objective (i.e., when R_i is close to zero). Only for an infinite length the two probabilities are the same, due to:

$$\lim_{len(S_i) \rightarrow \infty} \frac{R_i}{len(S_i)} = 1 . \quad (32)$$

Therefore, the SA tends to look at sequences that are likely to be always longer than the previous at any step. Besides, such SA diverges from any global optimum, because they are sequences of finite length.

Not only does α need to verify (24), but it should also be chosen carefully. In fact, the lower α is, the higher the average of the length of the sequences S_i will be. This happens because the weight of the length on the energy (21) decreases. Thus, the probability (9) of accepting a new longer sequence gets higher. The higher this average is, the more likely it is that the coverage will be greater. But at the same time the computational cost will increase as well.

In the former description, the SA has been studied with a fixed neighbourhood. However, it has been shown that the SA performs better if the neighbourhood size changes dynamically according to the temperature [41, 42]. The idea is to have a large size at the beginning of the search to boost the *exploration* of the search space. Then, the size should decrease to allow the *exploitation* of the current search region. It can be easily done if we consider, for getting the neighbourhoods of S_i , K_i operations on S_i . I.e., the new S_{i+1} will be generated using K_i operations on S_i instead of only one. Let K_0 be the initial size and N the total number of iterations of the search. We can write the size K_i at the iteration i as:

$$K_i = 1 + (k_0 - 1) \cdot \frac{N - i}{N} . \quad (33)$$

In that way, the neighbourhood size starts with a value of K_0 , and decreases uniformly until it arrives at 1.

3.4 Using Genetic Algorithms

Among the most used metaheuristics in software testing there are the Genetic Algorithms (GAs) [13]. To apply a GA for testing containers, we need to discuss:

Encoding, a chromosome can be viewed as sequences of FCs. Each single gene is a FC with its input parameters. The number of parameters depends on the particular method.

Crossover, it is used to generate new offspring combining between them the chromosomes of the parents. The offspring generated by a crossover are always well formatted test sequences. Therefore, no post processing is needed to adjust a sequence. The only particular thing to note is that the parents can have different lengths.

Mutations, they change an individual by a little. They are the same operations on a test sequence as described in sec. 3.2.

Fitness, the easiest way to define a fitness for the problem is Eq.(2). Anyway, we need to introduce the branch distance (5) in the fitness, otherwise the search will be blind:

$$f(S_i) = cov(S_i) + \frac{1}{2} \cdot (1 - B(S_i)) + \frac{1}{2} \cdot \frac{1}{1 + len(S_i)} . \quad (34)$$

Note that, for both the fitness, the following predicate is always true:

$$cov(S_i) > cov(S_j) \Rightarrow f(S_i) > f(S_j) \quad \forall S_i, S_j \in S . \quad (35)$$

Although the latter fitness gives good results, it can deceive the search in the same way as it happens in the Simulated Annealing. I.e., the use of the branch distance can lead to longer sequences without increasing the coverage in the end. To address this problem we can use a rank selection, but in a *stochastic* way [29]. I.e., for the selection phase, we can rank the individuals in the population using randomly either the fitness (2) or (34).

Container	Lines of Code	Functions under test	Best Reached Coverage	Max Coverage
Stack	118	5	10	10
Vector	1019	34	100	126
LinkedList	708	20	84	126
Hashtable	1060	18	106	213
TreeMap	1636	17	191	368
BinomialHeap	355	3	79	89
BinTree	154	3	37	37

Table 1: Characteristics of the containers in the test cluster. The highest coverages reached during the tests are reported.

Fitness sharing, it is used to maintain a degree of diversity inside the population. Individuals that have a chromosome close to others in the population will see their fitness decreasing. Common ways to define the distance between two individuals are the Euclidean and Hamming distances. When they are not suitable for a particular problem, a specific new distance measurement can be defined. Although the use of fitness sharing usually gives better results, it is not easy to define a right distance for this problem. In fact, the order of the FCs in the sequence is very important, and a single difference at the beginning of the sequence can completely change its behaviour. Besides, there can be a lot of read-only or redundant FCs that can be modified without altering the behaviour of the sequence. Therefore, in this work we did not use any sort of fitness sharing.

4 Case Study

To validate the novel techniques described in this paper, the following containers have been used for testing: `Vector`, `Stack`, `LinkedList`, `Hashtable` and `TreeMap` from the Java API 1.4, package `java.util`. On the other hand, `BinTree` and `BinomialHeap` have been taken from the examples in JPF [33]. Table 1 summarises their characteristics. The coverage values are referred to the branch coverage, but they also include the calls to the functions.

4.1 Comparing the Search Algorithms

The different algorithms described in this paper have been tested on the given cluster of containers. When an algorithm needs that some of its parameters should be set, experiments on their different values had been done. Anyway, these parameters are optimised on the entire cluster, and they remain the same when they are used on the different containers. Although different tests on these values have been carried out, there is no guarantee that the chosen values are the best. Random Search looks to sequences up to length 60. The cool rating in SA is set to 0.999, with initial neighbourhood size of 3. The value of α in (21) is 0.5. The GA uses a single point crossover with probability 0.2. Mutation probability of an individual is 0.9. Population size is 64. Rank selection is used with a bias of 1.5. Elitism rate is set to two individuals for generation.

Each algorithm has been stopped after evaluating 100,000 sequences. The machine used for the test was a Pentium with 3.0 GHz and 1024 M of ram running Linux. The tests have been run under a normal usage of the CPU. Table 2 reports the performances of these algorithms on 100 runs. Using the same data, a Mann Whitney U test [19] has been used to compare the performances of the different algorithms. The performance

Container	Search Algorithms	Coverage			Length		
		Mean	Variance	Median	Mean	Variance	Median
Stack	Random	10.00	0.00	10.00	6.74	0.43	7.00
	Hill Climbing	10.00	0.00	10.00	6.00	0.00	6.00
	Simulated Annealing	10.00	0.00	10.00	6.05	0.05	6.00
	Genetic Algorithm	10.00	0.00	10.00	6.00	0.00	6.00
Vector	Random	85.21	1.50	85.00	57.22	7.26	58.00
	Hill Climbing	99.91	0.08	100.00	52.61	56.11	50.00
	Simulated Annealing	99.94	0.06	100.00	45.23	0.89	45.00
	Genetic Algorithm	99.90	0.09	100.00	46.83	2.36	47.00
LinkedList	Random	70.03	2.25	70.00	54.73	23.49	56.00
	Hill Climbing	84.00	0.00	84.00	37.13	2.30	37.00
	Simulated Annealing	82.49	2.31	83.00	33.55	6.65	34.00
	Genetic Algorithm	83.76	0.35	84.00	36.97	4.27	37.00
Hashtable	Random	92.87	1.83	93.00	54.76	21.49	56.00
	Hill Climbing	106.00	0.00	106.00	35.04	0.03	35.00
	Simulated Annealing	105.86	0.72	106.00	35.00	0.76	35.00
	Genetic Algorithm	100.71	4.53	100.00	30.57	4.63	30.00
TreeMap	Random	151.97	8.55	152.00	53.81	31.99	55.00
	Hill Climbing	188.63	0.61	189.00	50.96	8.96	50.00
	Simulated Annealing	185.28	3.45	186.00	41.09	4.16	41.00
	Genetic Algorithm	185.22	4.19	185.50	42.46	7.64	42.00
BinomialHeap	Random	77.36	0.27	77.00	45.40	76.26	43.00
	Hill Climbing	78.12	0.55	78.00	25.15	35.90	27.00
	Simulated Annealing	76.33	0.22	76.00	16.31	36.15	14.00
	Genetic Algorithm	77.68	0.90	77.00	19.15	25.05	16.00
BinTree	Random	37.00	0.00	37.00	27.00	14.62	28.00
	Hill Climbing	37.00	0.00	37.00	9.12	0.10	9.00
	Simulated Annealing	37.00	0.00	37.00	9.32	0.32	9.00
	Genetic Algorithm	37.00	0.00	37.00	9.16	0.13	9.00

Table 2: Comparison of the different search algorithms on the container cluster. Each algorithm has been stopped after evaluating up to 100,000 solutions. The reported values are calculated on 100 runs of the test.

of a search algorithm is calculated using function (2). The level of significance is set to 0.05. HC and GA are statistically equivalent on **Stack** and **BinTree**. SA and GA are equivalent on **TreeMap**. If we compare table 2 with the best coverage ever reached (reported in table 1), we will see that only **TreeMap** is difficult to test. Besides, from that table the HC seems the best algorithm. That is a very interesting result, because usually local search algorithms are suggested of not being used for generating test data [37]. Although the search spaces for software testing are usually “complex, discontinuous, and non-linear” (J. Wegener et al. [37]), the evidences of our experiments lead us to say that it does not seem true for container classes. However, more tests on a bigger cluster of containers and the use of different search algorithms are required.

To increase the value of the techniques described in this paper, a comparison with other systems in the literature is required. Unfortunately, that was not possible to do. The main reason is that there is not a common set of classes that can be used as a benchmark. Although the API of Java is often used for getting the tested classes, how they are instrumented and which testability transformations have been applied are usually unknown. Even in the cases in which there is access to the original source codes, many technical problems arise.

4.2 Known limits

The system described in this paper is not able to generate input data for covering all the branch statements in the source code of the CuT. This is due to different reasons:

- The system also tries to cover the branches in the **private** methods. Anyway, the generated test sequences do not access directly to the **private** methods of the container. They can be executed only if at least one public method can generate a chain of function calls to

Container	Transformations
Stack	0
Vector	2
LinkedList	3
Hashtable	5
TreeMap	5
BinTree	0
BinomialHeap	0

Table 3: For each container in the tested cluster, the number of times that the novel transformation can be successfully applied is reported.

them. Although using the Java *reflection* a driver can directly call private methods of the CuT, besides the fact that a driver can be located inside the CuT, it has been preferred to test directly only the public methods. It is possible to do assumptions on the semantic of the public methods of a container (see section 2.2), but nothing can be said about the private ones. Thus, the proposed space reduction techniques cannot be applied to them.

- the very few public methods with odd input parameters (see section 2.2) cannot be directly called. If they are not called by other methods, they will not be tested.
- some methods can return objects which class is implemented inside the same file of the CuT or even in the same method. For example, the method `keySet` in `TreeMap` returns a `Set` which class is implemented inside `keySet`. Because the system does not call any method on the returned objects of the tested methods, such internal classes cannot be tested.
- for a given test sequence, only one constructor is called. That is not a difficult problem. To be fixed, it will require linking all not static function calls in a particular sequence to an available container instance. However, the search space will get larger.
Another option might be to use multiple sequences, each one that uses a different constructor.
- some branches can be infeasible. This is a general problem that is not related to the used testing system.

4.3 Effects of the Novel Testability Transformation

The use of the testability transformation described in section 2.3 has been investigated. Besides transforming the calls to `equals`, the semantic of `compareTo` is changed too. Table 3 shows the number of such method calls in the cluster of the container that take advantage from the transformation. Note that the implementations of the `BinTree` and the `BinomialTree` used do not handle objects but integer values. This is why the above methods do not appear in them. For `TreeMap`, it also has been considered the calls to the private method `compare`, because it includes and replaces the calls to `compareTo`.

Tests on the different containers have been carried out. However, no particular improvements on the performance of the algorithms have been

found. That can be explained by considering that the transformation is applied only to very few branches. Besides, these branches are “easy”. I.e., they are covered without any problem even if no guidance has been given to the search algorithms. However, that testability transformation gives no overhead to the computational cost of the search. Therefore, it should always be applied, because there might be containers that have difficult branches that depend on the `equals` method.

5 Related work

There have been different works focused on testing containers. Early works that use *testgraph analysis* can be found in [21], but they require a lot of efforts from the tester. In the same way, techniques that exhaustively search a container up to size N [28] cannot be applied to a new container without the help of the tester. In fact, the tester would be responsible for providing both the *generator* and the *test driver*.

However, a Java container is an Object Oriented (OO) program. Therefore, any tool that claims to automatically generate input data for a OO software should also work for a container. Different tools have been developed. The early ASTOOT [8] generates tests from algebraic specifications of the functions. Korat [4] and TestEra [20] use isomorphic generation of data structures, but need predicates that represent constraints on these data structures. Rostra[39] uses bounded exhaustive exploration with concrete values. On the other hand, tools that exploit the *symbolic execution* [15] are for example Symstra[40] and the work of Buy et al. [5]. In the same category, of particular interest there is the use of the model checker Java PathFinder (JPF) for test input generation [32]. Extensions of this approach, with a focus on containers, can be found in [33]. Although promising results have been given, these techniques are likely to not scale well. Besides, it is not clear how functions with objects as input can be handled when symbolic execution is used. In fact, it seems that all containers tested by these latter tools consider only the basic *int* type for the stored elements. Although [32] can consider objects as input, it needs to exploit the specification of the functions (in particular the precondition) to initialise such objects. It is important to outline that our system does not need neither any specification or any help from the user at all. It only needs to know that the CuT is a container.

Due to those limitations, the use of optimisation based techniques (e.g., Genetic Algorithms) for testing OO programs has started to be investigated in the last few years. To the best knowledge of the author, however, no work have been done specifically for testing containers. Tonella [30] used Genetic Algorithms (GAs) for generating unit tests of Java programs. Solutions are modelled like sequences of function calls with their inputs and caller (an object instance or a class if the method is static). Special new crossover and mutation operators are proposed to let each new generated solution feasible. Similar work with GAs has been done by Wappler and Lammermann [34], but they used standard evolutionary operators. That can cause the generation of infeasible individuals, which will be penalised by the used fitness function. Besides, they investigated the use of different layers of optimisation for target objects, methods and parameters. The use of Strongly Typed Genetic Programming for testing Java program has been done by Wappler and Wegener [36]. They extended this approach by considering the problem of the raised excep-

tions during the evaluation of a sequence [35]. If an exception is thrown, the fitness will consider how distant the method in which it is thrown is from the target method in the test sequence. Liu et al.[18] used a hybrid approach, in which Ant Colony Optimisation is exploited to optimise the sequence of function calls. Multi-agent Genetic Algorithm is used then to optimise the input parameters of those function calls. Also Cheon et al. [6] proposed an evolutionary tool, but they implemented and tested only a random search. It is important to highlight that, even if a testing tool is designed for handling a generic program, container classes are often used as benchmarks [39, 40, 32, 30, 34, 36, 18].

6 Conclusions

We presented optimisation based test data generation techniques for containers. To the best knowledge of the authors, this paper is the first that analyses this problem. Search algorithms like Random Search (RS), Hill Climbing (HC), Simulated Annealing (SA) and Genetic Algorithms (GAs) have been applied. Discussions on how and why a search can be deceived have been given, besides how to avoid it. Search space reductions and a novel testability transformation have also been presented. Although different settings of the algorithm parameters can lead to different results, it has been empirically shown that HC usually performs better than the other algorithms. That can seem strange, because local search algorithms are suggested of not being used for generating test data [37]. Although HC performs well on all the tested classes, the results on `TreeMap` are not completely satisfactory. That leads us to investigate new algorithms in the future for testing Java containers. A Memetic Algorithm [27] seems a good first candidate to be considered.

Besides presenting a working system, this paper wants to emphasise the need of comparing different search algorithms for a given software engineer problem. Too often, in the literature, there is a bias toward Genetic Algorithms. Although GAs have been applied successfully to a wide range of different problems, that does not necessarily mean that they will work well on a new problem [38]. The Search Based Software Engineering field is still too young for deciding what is the best search algorithm that should be applied to it.

The approach of trying to cover every branch at the same time with a single sequence is novel for search based testing. Therefore, its performance needs to be compared to the traditional approach of testing each branch separately. Future work will try to shed lights on which technique is better and why.

The discussed techniques have been applied to testing containers. If they are still valid and how to use them for testing generic OO software is a matter that needs to be investigated.

7 Acknowledgements

This work is supported by an EPSRC grant (EP/D052785/1).

References

- [1] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 43–52, 2004.
- [2] A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 2442–2454, 2003.
- [3] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1329–1336, 2002.
- [4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2002.
- [5] U. Buy, A. Orso, and M. Pezzè. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 39–48, 2000.
- [6] Y. Cheon, M. Y. Kim, and A. Perumandla. A complete automation of unit testing for java programs. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, pages 290–295, 2005.
- [7] M. L. Collard. Addressing source code using srcml. In *IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension (IWPC'05)*, 2005.
- [8] R. Doong and P. G. Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, pages 101–130, 1994.
- [9] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.
- [10] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1351–1358, 2002.
- [11] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.
- [12] M. Harman, M. Munro, L. Hu, and X. Zhang. Side-effect removal transformation. In *Proceedings of the 9th IEEE International Workshop on Program Comprehension*, pages 310–319, 2001.
- [13] J. H. Holland. *Adaptation in Natural and Artificial Systems, second edition*. MIT Press, Cambridge, 1992.
- [14] IEEE-Standards-Board. Ieee standard for software unit testing: An american national standard, ansi/ieee std 1008-1987. *IEEE Standards: Software Engineering, Volume Two: Process Standards*, 1999.
- [15] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, pages 385–394, 1976.

- [16] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.
- [17] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, pages 870–879, 1990.
- [18] X. Liu, B. Wang, and H. Liu. Evolutionary search in the context of object oriented programs. In *MIC2005: The Sixth Metaheuristics International Conference*, 2005.
- [19] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.
- [20] D. Marinov and S. Khurshid. Testera: A novel framework for testing java programs. In *IEEE International Conference on Automated Software Engineering (ASE)*.
- [21] J. McDonald, D. Hoffman, and P. Strooper. Programmatic testing of the standard template library containers. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 147–156, 1998.
- [22] P. McMinn. *Evolutionary Search for Test Data in the Presence of State Behaviour*. PhD thesis, University of Sheffield, 2005.
- [23] P. McMinn and D. Binkley. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the Third UK Software Testing Workshop*, pages 165–182, 2005.
- [24] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 2488–2500, 2003.
- [25] P. McMinn and M. Holcombe. Hybridizing evolutionary testing with the chaining approach. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1363–1374, 2004.
- [26] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1013–1020, 2005.
- [27] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech Concurrent Computation Program, C3P Report 826*, 1989.
- [28] P. Netisopakul, L. White, J. Morris, and D. Hoffman. Data coverage testing of programs for container classes. In *Proceedings 13th International Symposium on Software Reliability Engineering*, pages 183–194, 2002.
- [29] T. P. Runarsson and X. Yao. Stochastic ranking for constrained evolutionary optimization. *IEEE Transactions on Evolutionary Computation*, 4(3):284–294, 2000.
- [30] P. Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.
- [31] N. Tracey, J. Clark, K. Mander, and J. A. McDermid. An automated framework for structural test-data generation. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 285–288, 1998.

- [32] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004.
- [33] W. Visser, C. S. Pasareanu, and R. Pelànek. Test input generation for java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 37–48, 2006.
- [34] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1053–1060, 2005.
- [35] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In *Congress on Evolutionary Computation (CEC)*, pages 851–858, 2006.
- [36] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1925–1932, 2006.
- [37] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.
- [38] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.
- [39] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 196–205, 2004.
- [40] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, 2005.
- [41] X. Yao. Simulated annealing with extended neighbourhood. *International Journal of Computer Mathematics*, 40:169–189, 1991.
- [42] X. Yao. Comparison of different neighbourhood size in simulated annealing. In *Proceedings of the Fourth Australian Conf. on Neural Networks (ACNN'93)*, pages 216–219, 1993.