Hybridizing Evolutionary Testing with Artificial Immune Systems and Local Search

Konstantinos Liaskos and Marc Roper
Department of Computer and Information Sciences
University of Strathclyde, Glasgow, Scotland
{konstantinos.liaskos, marc.roper}@cis.strath.ac.uk

Abstract

Search-based test data generation has been a considerably active research field recently. Several local and global search approaches have been proposed, but the investigation of Artificial Immune System (AIS) algorithms has been extremely limited. Our earlier results from testing six Java classes, exploiting a Genetic Algorithm (GA) to measure data-flow coverage, helped us identify a number of problematic test scenarios. We subsequently proposed a novel approach for the utilization of Clonal Selection. This paper investigates whether the properties of this algorithm (memory, combination of local and global search) can be beneficial in our effort to address these problems, by presenting comparative experimental results from the utilization of a GA (combined with AIS and simple local search (LS)) to test the same classes. Our findings suggest that the hybridized approaches usually outperform the GA, and there are scenarios for which the hybridization with LS is more suited than the more sophisticated AIS algorithm.

1. Introduction

Search-based test case generation techniques, such as Evolutionary Testing (ET), are well researched for procedural software, but relatively little research has been done in the area of evolutionary object-oriented testing.

Object-orientation is one of the dominant paradigms in software engineering. Code reuse is a fundamental characteristic of this technology and at the same time one of its most important benefits. A class becomes the basic software unit that possesses attributes, constructors and methods. Thus, new challenges are introduced in the testing process, which lead to the realization that the problem of automatic test data generation needs to be re-examined in this new context. The authors have argued [12] that data-flow coverage is more suited under these circumstances, because it reflects more accurately the way that the methods manipulate the attributes of a class.

This paper extends our previous work [12] in the area of automatic test data generation. Our experimental results from testing six classes from the standard Java library using a GA were promising, but they also suggest that there are a number of problems that need to be tackled in order to achieve high levels of data-flow coverage.

In order to tackle these problems, we proposed [13] a novel approach for the utilization of an AIS algorithm (Clonal Selection in particular) and we demonstrated how the features introduced by it (i.e. immunological memory using memory cells, combination of local and global search) can be beneficial in our effort to address the problematic test targets identified by the GA.

The ultimate goal of this paper is to investigate this by hybridizing a GA with the Clonal Selection algorithm in order to test the same six Java classes and compare the results against those obtained when the GA alone was used. Additionally, we look into the case when the GA is combined with a much simpler local search algorithm.

The results provide evidence to support that the performance of the GA alone is usually outperformed by the hybridized algorithms, and that surprisingly there exist test scenarios for which the hybridization with LS is more suited than the more sophisticated Clonal Selection algorithm.

The major contributions of this paper can be summarized as follows:

- It presents a framework for the application of AIS algorithms to the problem of test data generation.
It proposes the hybridization of GAs with the Clonal Selection algorithm and a local search algorithm.

It provides a comparative study of the hybrid approaches.

2. Previous work & motivation

The key research component of our previous [12] work was to advance evolutionary techniques to generate test data for object-oriented software using data-flow as a coverage criterion.

2.1. Data-flow coverage

One issue of great importance is which coverage criterion to adopt, since it can affect in several ways the design of the search technique in use. For example, the design of the fitness function depends on the adopted coverage criterion. Cornett [4] reports that data-flow coverage has the advantage of requiring very thorough testing because the paths reported have direct relevance to the way the program handles data. In this context, data-flow coverage is particularly appropriate for object-oriented testing, where classes combine data and behavior in the form of methods that access and manipulate that data.

Two drawbacks of this coverage criterion are the high complexity (although the large number of paths is significantly reduced compared with path coverage) and the fact that it does not include decision coverage.

In the case of testing procedural programs a procedure is considered in isolation and the set of test requirements consists of individual statements, branches, etc., depending on the adopted coverage criterion. Even in the case of object-oriented testing the test requirements (branches) are defined and examined for each method individually. Thus, although the resulting test cases consist of arbitrary sequences of method invocations, the last method invoked must be the method currently under test. In the case of data-flow coverage, things are more complicated, because a test requirement is not a simple statement, branch or condition, but a def-use (d-u) pair. In brief, these are ordered pairs of mutually related statements in which the first statement defines and the second statement uses the same variable. As a result, a test target with respect to a variable consists of the def-statement and the use-statement for this variable (provided that there is at least one definition-clear path between def and use).

Harrold and Rothermel [9] present an approach that supports data-flow testing in a class. For individual methods, and methods that send messages to others within the class, their technique is similar to existing data flow techniques ([3, 10, 11, and 17]). For methods that are accessible outside the class and can be called in any order by the users of the class, they propose a method to compute data-flow information and use it to test possible interactions between these methods. The authors define three types of d-u pairs in classes that must be tested: intra-method, inter-method and intra-class d-u pairs. According to this classification, three levels of data-flow testing of classes can be defined:

*Intra-method testing*: Every public method of the CUT is tested in isolation, which means that the intra-method d-u pairs for every individual method are tested. Thus, the number of test targets that the GA has to satisfy is finite.

*Inter-method testing*: Each public method of the CUT is tested together with other methods in the class that it calls directly or indirectly. Existing frameworks ([3, 10, 11, and 17]) can be utilized to compute the finite set of inter-method d-u pairs of the CUT.

*Intra-class testing*: The interactions of public methods of the class under test when they are called in arbitrary sequences are tested. The major difficulty in this case is that the number of possible method invocations in order to test the d-u pairs that fall under this category is infinite. Thus, only a subset of all these sequences can be computed.

2.2. Evolutionary testing of classes

Our initial study focused on the design of the fitness function: every statement of the d-u pair is considered a partial target, for which traditional fitness computation is utilized, the representation of the candidate solutions and the implementation of the GA. When a critical branch for each partial test target is missed, the fitness value of each candidate solution is computed as the sum of a measure of closeness to the test target and the degree of closeness to satisfying the condition that led to the miss (fitness = approximation_level + normalised(branch_distance) [1]). The candidate solutions are encoded into chromosomes comprising a sequence of constructor and method invocations and a set of input values (passed as parameters to the invoked methods). The class under test is analyzed prior to the execution of the GA in order to identify the test targets (d-u pairs). JUnit is used to execute the resulting test cases, and various coverage tools are used to measure different types of coverage.

2.3. Initial experimental results (GA)
Six classes from the standard Java library (JDK 5.0) were used for our experiments. Table 1 summarizes the features of the six classes under test (for more details refer to Java documentation). Based on the average size of each class and the average number of methods, it is safe to say that all the six classes represent a good sample of the java.util package. Furthermore, the internal complexity of the classes varies and is representational of the standard Java library.

Table 1. Summary of the features of the classes under test

<table>
<thead>
<tr>
<th>Class under test</th>
<th>Lines of code</th>
<th>Public methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>118</td>
<td>5</td>
</tr>
<tr>
<td>LinkedList</td>
<td>414</td>
<td>6</td>
</tr>
<tr>
<td>TreeSet</td>
<td>496</td>
<td>15</td>
</tr>
<tr>
<td>LinkedList</td>
<td>778</td>
<td>28</td>
</tr>
<tr>
<td>HashMap</td>
<td>1021</td>
<td>13</td>
</tr>
<tr>
<td>BitSet</td>
<td>1046</td>
<td>26</td>
</tr>
<tr>
<td>Average</td>
<td>&lt;646</td>
<td>&lt;18</td>
</tr>
</tbody>
</table>

The experiments were conducted on a PC (MS Windows XP, Home Edition, Version 2002, SP 1) with an Intel Pentium 4 3.00 GHz processor and 512 MB of RAM. Initially, the main parameters of the GA <populationSize, maxAttempts, maxExecutionTime> had the following values: <40, 20, 600s>. In the cases that the achieved level of code coverage was not satisfactory (TreeSet, LinkedList, HashMap and BitSet), the parameter values were changed to: <40, 30, 3600s>. In the case of the BitSet class we had to use the values: <50, 40, 5400s> in order to obtain a satisfactory level of code coverage.

Based on the fact that the ultimate goal of our testing approach is to achieve high levels of code coverage, we executed the resulting test cases to measure definition-use coverage, branch coverage and statement coverage. We used the tool Coverlipse to obtain definition-use coverage, which is the adopted coverage criterion by our approach. Because of the fact that Coverlipse can compute only intra-method def-use coverage, we had to implement a mechanism that computes inter-method and intra-class coverage (as the number of the covered test targets divided by the total number of test targets). The tool DJUnit was used to obtain branch coverage. Statement coverage is provided by both Coverlipse and DJUnit. Table 2 contains a summary of the experimental results.

<table>
<thead>
<tr>
<th>Class under test</th>
<th>Definition-use coverage</th>
<th>Branch coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>LinkedList</td>
<td>99%</td>
<td>98%</td>
</tr>
<tr>
<td>TreeSet</td>
<td>90%</td>
<td>92%</td>
</tr>
<tr>
<td>LinkedList</td>
<td>90%</td>
<td>98%</td>
</tr>
<tr>
<td>HashMap</td>
<td>90%</td>
<td>98%</td>
</tr>
<tr>
<td>BitSet</td>
<td>92%</td>
<td>98%</td>
</tr>
<tr>
<td>Average</td>
<td>95.5%</td>
<td>94.8%</td>
</tr>
</tbody>
</table>

2.4. Problematic test scenarios

Statement, branch and d-u coverage were measured and compared with each other and with the branch coverage from [18] that uses the same classes as test objects. D-u coverage is relative high, but lower than branch or statement coverage in the case of classes with high cyclomatic complexity. Lower coverage, higher computational cost, and the large number of test cases can be explained by the fact that the number of test goals in our case is much higher compared to branch coverage. The investigation of our experimental results helped us to identify a number of problematic test scenarios and classify the uncovered test targets into three categories: equivalent, subsequent, and unrealistic test targets. Figure 1 illustrates that with three code examples, and a brief description for each category follows.

Equivalent test targets: The bitIndex parameter and the local variable unitIndex form one d-u pair each. The latter was not covered during our experiment. These two test targets are classified as equivalent (i.e.,
they correspond to the same code structure). If we take into account this fact during the analysis of the CUT, the amount of the test targets is reduced by 50%. It is obvious that this situation introduces an additional overhead to the overall execution time of GA. It is worth noting that the bitIndex parameter may affect other d-u pairs as well (deriving from calls from other public methods of the BitSet class or by passing an instance field as a parameter to the flip method). Thus, a good remedy in this case would be to take into account the d-u pair formed by the bitIndex parameter and ignore the other one. The same approach could be followed in the case that we had a local variable and an instance field forming equivalent test targets. However, this would not be so straightforward if we had a parameter and an instance field forming equivalent d-u pairs. In this case, the approach to be followed depends on every individual test target.

Subsequent test targets: In the example above (clear method in BitSet), the d-u pairs formed by the parameter fromIndex and the local variable startUnitIndex are consequent test targets (i.e., the satisfaction of one of them is strongly related with the other). When the startUnitIndex d-u pair is satisfied, we are very close to satisfying the fromIndex d-u pair. Our GA deals with every test target individually. Despite the fact that the first target was covered, the second was not, because the GA started from scratch to cover it. Thus, valuable effort and information go to waste.

Unrealistic test targets: The line where the local variable targetCapacity is redefined within the if-statement is very difficult to reach. According to all the generated test cases, a vast number of elements must be inserted into the hash table in order for this fragment of code to be executed. This situation can be considered unrealistic, as are all the d-u pairs derived from this definition.

3. Proposed solution

As a means to address these problematic test scenarios, the investigation of AIS algorithms was proposed [13]. The human immune system has remarkable immune properties: learning and adaptation implemented by affinity maturation (figure 2), immunological memory using memory cells, self/non-self discrimination by negative selection. AIS were defined as computational systems inspired by theoretical immunology and observed immune functions, principles and models, applied to solve problems. This paper investigates how these features can be beneficial in our effort to improve the performance of our previously implemented GA.

3.1. Human immune system

The human body is continuously under attack from pathogens such as bacteria and viruses coming through digestive systems or damaged tissues. The immune system in our body performs several functions to protect the organism against the pathogens. To do so, there are two aspects to the problem that the immune system faces. The primary problem is the identification or detection of the pathogens. After recognizing a disease, the efficient elimination of those pathogens is the next responsibility of the immune system.

The identification problem is often described as that of distinguishing the body’s own tissues from the pathogens. In this case, the body’s own tissues are called self while pathogens are called non-self, and the process of distinguishing between self and non-self antigens (Ag) is termed self/non-self discrimination. Once pathogens have been identified, they must be eliminated by the immune system. The components of the immune system which accomplish this are called effectors. In other words, the immune system should select the right effectors to eliminate the various pathogens.

There are several types of immune cells (refer to [5, 7, and 8] for an in-depth description), but our work will focus on the lymphocytes which are white blood cells. B-cells (figure 3) and T-cells are main types of lymphocytes and play a role to detect pathogens. These cells are generated and developed in two organs such as the bone marrow and the thymus. Both B-cells and T-cells are generated in the bone marrow, but only B-cells are developed within the bone marrow. On the other hand, T-cells migrate to the thymus and mature. Both cell types have receptor molecules on their surfaces to identify the antigenic patterns of pathogens. The T-cell receptor is called TCR and the B-cell receptor is called BCR or antibody (Ab).
3.2. Artificial immune systems (AIS)

In order to apply AIS to real-world problems, it is critical to address the following questions:
- How do we mathematically represent immune cells and molecules?
- How do we quantify their interactions or recognition?
- How do we form the procedures of the variety of the observed functions in the human immune system?

The answers to these questions will result in a generic framework for the application of AIS techniques to solve real-world computational problems. De Castro et al. [7] present a taxonomy of AIS algorithms. The focus of our research is on population-based algorithms (a discussion of the network-based algorithms is for the moment out of the scope of this paper):

**Clonal Selection algorithm:** In order to eliminate the pathogens which grow exponentially in our body, the immune system reproduces those cells capable of identifying and binding with pathogens using a cloning method. An algorithm named CLONALG [6] for clonal expansion was developed. The cells are selected and cloned proportionally to their affinity. In contrast, the cells are mutated inversely proportionally to their affinity. Figure 4 provides a high-level description of the main steps of the algorithm.

- Generate a set of N candidate solutions (antibody repertoire) in a shape-space to be defined by the problem under study;
- repeat
  - Select n highest affinity cells in relation to the antigen set to be recognized or to the function being optimized;
  - Clone (generate identical copies of) these n selected cells. The number of copies is proportional to their affinities: the higher the affinity, the larger the clone size (number of offsprings);
  - Mutate with high rates (hypermutation) these n selected cells with a rate inversely proportional to their affinities: the higher the affinity, the smaller the mutation rate;
  - Re-select n1 highest affinity mutated clones to compose the new repertoire;
  - Replace some low affinity cells by new ones;
- until a given stopping criterion is met.

**Negative Selection algorithm:** T-cells migrate to and mature within the thymus. When the T-cells enter the thymus, they are selected negatively by the thymus. Those cells which recognize self antigens are eliminated and all T-cells which do not recognize the self antigens survive in the thymus.

4. Clonal Selection algorithm vs. GAs

As described in the previous section, the Clonal Selection algorithm appears to be evolutionary-like. It is a population-based algorithm that employs mutation to generate new populations, and has a selection mechanism under which solutions from the current population are selected to survive (proportionally to their fitness). In both cases the candidate solution must be encoded. Unlike GAs, the Clonal Selection algorithm performs mutation on the selected solutions with a rate that is inversely proportionate to their fitness, and does not use crossover. The Clonal Selection algorithm introduces the notion of memory (i.e. high quality solutions are stored for future use, leading to a faster immune response). Despite the fact that a generic version of the Clonal Selection algorithm does not retain a fixed population size, a modified version can be used where the population size is not variable. In this way a problem can be solved using a minimal amount of resources (by having a high quality and economical set of solutions). A second look at figure 4 makes quite obvious the fact that in the case of the Clonal Selection algorithm there are more parameters that need to be fine-tuned. In that respect, the algorithm introduces a higher level of complexity and sophistication in comparison with a GA. Figure 5 summarises these parameters.

5. AIS in the context of automated testing

In this section we describe a framework (by addressing the questions stated in Section 3.2) that can be utilized to solve the problem of automatic test case generation. For a more detailed description with examples, please refer to our previous publication [13].
5.1. Representation

B-cells and T-cells can be represented by the encoded test cases. An encoding identical to the one used by the GAs can be used (test-cases encoded as chromosomes). No distinction between B-cells and T-cells is required (B-cells in the case of Clonal Selection and T-cells that have the ability to recognise B-cells in the case of Negative Selection).

Similarly to the case of GAs, the candidate solutions (test cases) need to be executed in order to record the executed path of the unit under test. Thus, the receptors of the immune cells can be represented as encoded versions of the executed paths. Following the same analogy, antigens can be represented as the test targets, which in turn are encoded representations of the code structure that we want to be covered by our test data. This representation results in a metaphor where immune cells (encoded test cases) have receptors (recorded paths from the execution of the test cases) that try to recognise antigens (encoded test targets).

In all cases, a binary encoding can be used. In the case of path coverage for instance, immune cell receptors can be encoded into binary strings (1’s for the executed edges of the control flow graph (CFG) and 0’s for those not executed). Similarly, antigens (test targets) can be encoded as binary strings of the desired path in the CFG. In the case of branch coverage, an analogous approach can be followed (for example encode receptors and antigens as bitstrings of 1’s when a branch is evaluated to true and 0’s when a branch is evaluated to false). This is the approach used in our implementation, since we are interested in the critical branches (i.e. branches that if not satisfied lead away from the test target) of the execution paths in order to compute the affinity.

5.2. Affinity computation

As proposed in the previous section, a binary representation of immune cell receptors and antigens makes it possible for a binary affinity measure to be utilised. The degree of matching between a B-cell receptor and an antigen can be expressed as a binary distance of the two strings. Similarly to evolutionary testing (trying to minimise the fitness function), our goal is to minimise the binary distance between receptors and antigens. It is worth noticing that the affinity measure takes into account all the partially matching sub-paths of the CFG. This is not the case in the traditional way that the fitness in the case of GAs is computed (the fitness value is computed at the first non-matching edge of the executed path and the rest of it is ignored). This feature combined with the “memory” aspect of the clonal selection algorithm and the fact that mutation is applied with a rate that is inversely proportionate to the affinity value make AIS techniques very promising for the problem of automatic test case generation.

By applying mutation inversely proportionate to affinity the Clonal Selection algorithm allows the exploration of local areas around an antibody by making small steps towards an antibody with higher affinity, leading to local optima. Because mutations with lower affinity are lost, the antibodies tend to go up the hill. At the same time, the introduction of “fresh” antibodies (usually randomly generated) helps to preserve the diversity of the population and prevents the algorithm from getting stuck in local optima (this helps the algorithm to take large steps through the landscape and possibly land on a location where the affinity is lower, but such a leap might lead to new areas of the search space previously unexplored and to antibodies on the side of a hill where the climbing region is more promising, and eventually to reach a global optimum).

To relate this with our previous work, this feature can be beneficial to tackle the problem of consequent test targets (reminder: the satisfaction of a test target is strongly related with another), in which case all is usually needed is a slight mutation of the test case that cover one target in order to cover the other one. This would be more difficult in evolutionary testing, since the mutation rate is fixed and if one takes place may lead the search away from covering the consequent test target. Under this perspective, the utilization of the Clonal Selection algorithm can be very useful for d-u and path coverage, in which cases a condition results in multiple d-u pairs and paths respectively.

The memory feature of the AIS algorithms appears even more promising to tackle the problem of consequent test targets. The fact that an antibody with high affinity is placed into the memory set provides a good starting point for the new search, something that would not happen with a traditional evolutionary algorithm (it would start with a fresh randomly generated population).

More importantly, AIS memory can be very useful to tackle the equivalent test targets problem (reminder: d-u pairs that correspond to the same code structure). An antibody represents a unique path of the CFG, but it may correspond to more than one d-u pairs (based on our experience and our results it usually does). By placing a clone of this antibody into the memory set, any d-u pair examined in a subsequent search that
5.3. Combining AIS and GAs

The reason we proposed the framework for the utilization of the Clonal Selection algorithm is not to advocate on favour of the AIS algorithms against GAs. The use of GAs in our previous work motivated the investigation of AIS. Our experimental results (despite being promising) suggested that there are some problems with evolutionary testing, which seem to be strongly related to the adopted coverage criterion and the fact that our approach is targeted on object-oriented software. The investigation of the AIS algorithms was chosen as a means to tackle these problems. Our ultimate goal and the main objective of this paper are to investigate whether it is possible to combine GAs and AIS algorithms to solve the problem of automatic test data generation. The latter are expected to introduce a high computational cost, but they also offer some features (as discussed in the previous section) that can be beneficial to achieve higher levels of d-u coverage. Our approach can be summarized as follows:

- use a traditional GA to automatically generate test-data
- for those test targets that are hard to cover, use an AIS algorithm, which can lead the search to more promising areas of the search space. The local search and memory features of the AIS algorithm are likely to result in higher d-u coverage. In an ideal situation, this step would be automated and the Clonal Selection would automatically take over from the GA, but in our current implementation we manually feed the algorithm with the test cases from the GA and then start its execution.

6. Local search

Since we argue that the attribute of combination of local and global search introduced by the Clonal Selection algorithm can be useful in our effort to improve the performance of the GA, we believe that also the hybridization of the GA with a local search algorithm must be investigated. This will provide evidence about whether it is worthwhile exploiting the more sophisticated and highly-complex (in terms of computational cost, parameter tuning and implementation cost) AIS algorithm instead of a much simpler local search algorithm.

Local search is a metaheuristic for solving computationally hard optimization problems. It can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions. Local search algorithms move from solution to solution in the space of candidate solutions (the search space) until a solution deemed optimal is found or a termination criterion is met. Figure 6 depicts the main steps of the algorithm. The way that the algorithm is used in our implementation can be summarized as follows:

- Populations of candidate solutions generated by the GA are used to initialise the LS algorithm.
- Although the main approach to avoid getting stuck in local optima is several restarts of the LS algorithm, we have not incorporated that in our implementation. In a different case, after the first restart all the information from the GA would be lost.
- Under the object-oriented paradigm, during the application of the alternating variable method and for each chromosome (encoded sequence of constructor and method invocation, and parameter values) the neighbourhood is explored as follows:
  - the method is applied firstly to the parameters of the invoked methods (starting with those that contains the definition and use)
  - if no further improvement occurs, the method is then applied to the parameters of the invoked constructors.

The latter is illustrated with the following example. Considering the test case of this example and assuming that the test target contained in method `meth` is missed, the alternating variable method is applied to the numerical parameters of `meth` until the target is covered or no further improvement is achieved. In the latter case, the method is applied to the integer parameter of the constructor `CUT`. One could argue that the
invocation of the other method of class CUT should be part of the neighbourhood exploration. Additionally, in the case of objects passed as parameters (obj of type A of meth in our example) these should be included in the neighbourhood exploration (e.g., by applying the alternating variable method to class A in the same way we do for CUT).

```java
//class under test
class CUT{
    public CUT(){
    }
    public CUT(int par1){
    }
    //method containing the test goal
    public int meth(A obj, int par2, int par3){
    }
}

public class A {
    public A(){
    }
    public void meth1(){
    }
    public int meth2(int par1, int par2){
    }
}
```

Test case:
```
CUT cut = new CUT(3);
a = new A();
a.meth2(4, 6);
cut.meth(a, 9, 6);
```

```
start with a randomly selected solution
repeat
    investigate neighboring solutions
    if fitness function is improved
        replace solution
until no further improvement for objective function
```

Figure 6. Local search algorithm

7. Experimental evaluation

7.1. Experimental results (GA + AIS, GA + LS)

These experiments were conducted on the same system described in Section 2.3. The parameters from figure 5 of the Clonal Selection algorithm were set as follows: N (size of the Ab repertoire) = 40, m (size of the memory set) = 30, r (remaining Ab repertoire) = 10, d (set of d lowest affinity Ab’s that will be replaced by new individuals) = 5, N_{gen} (maximum number of generations) = 150, n (number of highest affinity individuals to be chosen) = 20, \( \beta \) (multiplying factor for the total number of clones) = 1. Tables 3 and 4 contain the experimental results (with the coverage achieved by the GA alone in the parentheses).

<table>
<thead>
<tr>
<th>Class under test</th>
<th>d-u coverage intra-method (%)</th>
<th>d-u coverage all types (%)</th>
<th>Branch coverage (%)</th>
<th>Statement coverage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>(100)100</td>
<td>(100)100</td>
<td>(100)100</td>
<td>(100)100</td>
</tr>
<tr>
<td>StringTokenizer</td>
<td>(99)100</td>
<td>(98)100</td>
<td>(100)100</td>
<td>(100)100</td>
</tr>
<tr>
<td>TreeSet</td>
<td>(93)98</td>
<td>(93)97</td>
<td>(96)98</td>
<td>(97)98</td>
</tr>
<tr>
<td>LinkedList</td>
<td>(96)98</td>
<td>(94)96</td>
<td>(96)98</td>
<td>(97)98</td>
</tr>
<tr>
<td>HashMap</td>
<td>(94)97</td>
<td>(93)96</td>
<td>(94)96</td>
<td>(98)98</td>
</tr>
<tr>
<td>BitSet</td>
<td>(91)96</td>
<td>(90)96</td>
<td>(95)96</td>
<td>(97)97</td>
</tr>
<tr>
<td>Average</td>
<td>(95.5)98.2</td>
<td>(94.7)97.8</td>
<td>(96.8)98.5</td>
<td>(98.98)5</td>
</tr>
</tbody>
</table>

Table 3. Test coverage for the GA+AIS hybrid

Table 4. Test coverage for the GA+LS hybrid

7.2. Analysis of the results & findings

A first look at tables 2, 3 and 4 makes quite clear the fact that the hybridized approaches outperform the GA alone. All types of code coverage show a significant improvement when the GA was combined with the Clonal Selection algorithm. The scale of this improvement is such, that we can use it as evidence to support our initial hypothesis. The novel properties introduced by the AIS algorithms were beneficial in our effort to improve the code coverage of the GA approach. In the case that the GA is combined with the local search algorithm, the results are again promising (code coverage is improved or in the worst case scenario is as good as the GA approach). In the rest of this section we attempt to provide an insight of our experimental results by focusing on specific test scenarios.
The memory attribute of Clonal Selection is proved to be the most beneficial. It is worth mentioning that all the d-u pairs categorized as subsequent test targets in Section 2.4 were covered (38/38), when the GA was combined with the AIS algorithm (22/38 were covered by the GA and the remaining 16 were covered by the hybrid). Taking into account the fact that the Clonal Selection algorithm performed as well when run alone (these results were omitted for space saving reasons), we have satisfactory evidence to argue that the problem of the equivalent d-u targets was successfully tackled.

The memory property has also contributed to our effort to address the d-u pair categorized as subsequent test targets in Section 2.4, by saving test cases with high affinity for future generations. However, the major contribution toward this end was made by the local search element introduced by the Clonal Selection algorithm. Despite the fact that not all of the test targets under this category were covered (11/41 were covered by the GA and 17 out of the 30 remaining were covered by the hybrid), the improvement is still significant.

It is worth mentioning that test targets under the category unrealistic (see Section 2.4) were covered by neither of the hybrid approaches.

The results from the hybridization of the GA with the local search algorithm are promising as well. They usually show an improvement in terms of code coverage or at least same coverage as the GA alone. Although no category of problematic test scenarios was tackled as effectively as in the case of the hybridization with Clonal Selection, the algorithm performed as well for the subsequent test targets. Taking this into account and also the fact that in some cases there is not a significant difference in terms of code coverage, especially for those classes with low cyclomatic complexity (the classes in tables 2, 3, and 4 are in ascending order in terms of their cyclomatic complexity), we can say that there exist test scenarios in which the simple local search algorithm is more suited than the more sophisticated and highly-complex Clonal Selection algorithm.

However, the lack of memory suggests that the LS algorithm is not suitable to address the problematic test targets categorized as equivalent. Another possible reason why the levels of code coverage from hybridization with the LS are lower could be the fact that the paradigm of object orientation introduces a number of restrictions (as illustrated by the example in Section 6) that we should probably look into more thoroughly in the future in order to find a more effective way to explore the neighbourhood of candidate solutions.

8. Related work

McMinn [15] presents an extended survey of the application of metaheuristic search techniques (including GAs and local search) to the problem of automatic test case generation.

McMinn et al. [16] present the first theoretical analysis that tries to answer the question: “When and why Evolutionary Testing works?” In order to support the theory and compare the performance of ET to other search techniques, the authors present an empirical study that considers real world programs.

Baudry et al. [2] propose mutation analysis (faults are injected in the original software component and then the number of faults detected by the test cases is measured) as a means to measure the quality of test sets. Their study focuses on the problem of test case optimization and two approaches are presented: one using a GA and another one that uses an AIS algorithm (bacteriological approach). They report experimental results from a case study for C# programs which suggest that the quality of the initial test sets was improved significantly (mutation scores increased from 60% up to 90%), and also that the bacteriological approach outperformed the one using the GA.

May et al. [14] propose the utilization of AIS algorithms for an adaptive mutation testing system as a means to address the problem of high generation and execution time of mutant programs in the mutation testing domain. Their approach focuses on the creation of a system that monitors software development in order to generate a vaccine (a high quality test set that consists of a minimal amount of well-targeted test cases). Despite the fact that no experimental results are presented, they argue that the application of AIS to mutation testing is a plausible concept.

9. Conclusions and future work

This paper presents our most recent research in the area of automated test data generation for object-oriented software. The paper started by summarizing our previously published research [12] from testing six classes from the standard Java library in order to demonstrate our motivation and identify the problem that we attempt to tackle. The application of a GA to the problem of automatic test data generation for data-flow coverage revealed a number of problematic test scenarios. The utilization of Clonal Selection (one of the most popular population-based AIS algorithms) is proposed as a means to address these problems. A framework that maps this algorithm to the problem of
automatic test data generation was also presented. The main contribution of this paper is the hybridization of the GA with the Clonal Selection algorithm and a much simpler local search algorithm. The experimental results from the utilization of the two hybrid approaches to test the same six Java classes were presented. Our findings provide evidence to support that the performance of the GA alone is usually outperformed by the hybridized approaches, and that surprisingly there exist test scenarios for which the hybridization with LS is more suited than the more sophisticated Clonal Selection algorithm.

In the near future, our research is going to focus on further investigation of the proposed hybrid techniques. The next step will be the experimentation with larger-scale, real-world systems as test objects. This will provide more evidence to support our hypothesis that the combination of GAs with AIS algorithms can be beneficial for the problem of automatic test data generation. We also intend to have a more thorough look into the local search algorithm under the OO generation. We also intend to have a more thorough investigation of the proposed hybrid techniques.

The next step will be the experimentation with larger-scale, real-world systems as test objects. This will provide more evidence to support our hypothesis that the combination of GAs with AIS algorithms can be beneficial for the problem of automatic test data generation. We also intend to have a more thorough look into the local search algorithm under the OO paradigm and investigate ways to improve its performance.

10. References


