A Strategy for Evaluating Feasible and Unfeasible Test Cases for the Evolutionary Testing of Object-Oriented Software

José Ribeiro\textsuperscript{1}  Mário Zenha-Rela\textsuperscript{2}  Francisco Vega\textsuperscript{3}

\textsuperscript{1}Polytechnic Institute of Leiria, Portugal
\textsuperscript{2}University of Coimbra, Portugal
\textsuperscript{3}University of Extremadura, Spain

Automation of Software Test, 2008
A Strategy for Evaluating Feasible and Unfeasible Test Cases for the Evolutionary Testing of Object-Oriented Software

Outline

Introduction
  Software Testing
  Evolutionary Algorithms
  Evolutionary Testing

Approach and Methodology

Test Case Evaluation
  Numerical Formulation of the Test Goal
  Feasible and Unfeasible Test Cases
  Weight Reevaluation

Experimental Studies
  Probabilities of Operators
  Evaluation Parameters
  Discussion

Conclusions and Future Work
A Strategy for Evaluating Feasible and Unfeasible Test Cases for the Evolutionary Testing of Object-Oriented Software

- **Introduction**

**Outline**

- **Introduction**
  - Software Testing
  - Evolutionary Algorithms
  - Evolutionary Testing

- **Approach and Methodology**

- **Test Case Evaluation**
  - Numerical Formulation of the Test Goal
  - Feasible and Unfeasible Test Cases
  - Weight Reevaluation

- **Experimental Studies**
  - Probabilities of Operators
  - Evaluation Parameters

- **Discussion**

- **Conclusions and Future Work**
A Strategy for Evaluating Feasible and Unfeasible Test Cases for the Evolutionary Testing of Object-Oriented Software

Introduction

Software Testing

Test data selection, generation and optimization deals with locating good test data for a particular test criterion.

However, locating quality test data can be time consuming, difficult and expensive.

Test Data Generation

Automating the test data generation process is vital to advance the state-of-the-art in software testing.
Evolutionary Algorithms

- Evolutionary Algorithms use simulated evolution as a search strategy to evolve candidate solutions, using operators inspired by genetics and natural selection.
- The best known algorithms in this class include:
  - Evolution Strategies,
  - Evolutionary Programming,
  - Genetic Algorithms, and
  - Genetic Programming.

- Traditional **evolutionary operators** include:
  - Reproduction,
  - Mutation, and
  - Crossover.
Genetic Programming

- Genetic Programming is a machine-learning approach usually associated with the evolution of tree structures.
- It focuses on automatically creating computer programs by means of evolution.

Strongly Typed Genetic Programming (STGP)

- Was proposed with the intention of addressing the “closure” limitation of the Genetic Programming technique.
- Is particularly suited for representing method call sequences in strongly-typed languages such as Java, as it enables the reduction of the search space to the set of compilable sequences.
Evolutionary Testing

ET and SBTCG

- The application of evolutionary algorithms to test data generation is often referred to as *Evolutionary Testing* or *Search-Based Test Case Generation*.

- The **problem** is finding a set of input data (test cases) that satisfies a certain test criterion.

- The **search-space** is the input domain of the test object.

- Evolutionary Testing is an emerging technology for automatically generating high quality test data.
A Strategy for Evaluating Feasible and Unfeasible Test Cases for the Evolutionary Testing of Object-Oriented Software

Introduction

Evolutionary Testing

Example Test Case

A a = new A();
B b = new B();
b.f(2);
a.m(5, b);

- **Method Under Test:** m
- **Test Cluster:** A, B
- **Invocation of f on b aims at changing the state of b before passing it to m.**

The State Problem

- Occurs with objects that exhibit state-like qualities by storing information in fields that are protected from external manipulation – **encapsulation.**
Outline

Introduction
  Software Testing
  Evolutionary Algorithms
  Evolutionary Testing

Approach and Methodology

Test Case Evaluation
  Numerical Formulation of the Test Goal
  Feasible and Unfeasible Test Cases
  Weight Reevaluation

Experimental Studies
  Probabilities of Operators
  Evaluation Parameters
  Discussion

Conclusions and Future Work
Our Approach

- The focus of our on-going work is that of employing evolutionary algorithms for generating and evolving test cases for the structural unit-testing of third-party object-oriented Java programs.

- The main goal is to develop an automated test case generation tool – eCrash.
Our Approach

- Test cases are evolved using the Strongly-Typed Genetic Programming technique.

- Test data quality evaluation includes instrumenting the test object, executing it with the generated test cases, and tracing the structures traversed in order to derive coverage metrics.

- The strategy for efficiently guiding the search process towards achieving full structural coverage involves favouring test cases that exercise problematic structures and control-flow paths.

- Static analysis and instrumentation is performed solely with basis on the information extracted from the test objects’ Java Bytecode.
Test Case Representation

Controller controller0 = new Controller();
Controller controller1 = new Controller();
Config config2 = controller1.getConfig();
controller0.reconfigure(config2);
Controller controller3 = new Controller();
Config config4 = controller3.getConfig();
int int5 = 4;
config4.setPort(int5);
int int6 = 7999;
config4.addSignal(int6);
controller0.reconfigure(config4);

Figure: STGP tree and the corresponding Method Call Sequence.
Test Object Representation

```java
public void reconfigure(Config cfg)
0:  aload_1
1:  invokevirtual cfg.Config.getSignalCount ()I (6)
4:  iconst_5
5:  if_icmpne #16
8:  new java.lang.Exception (7)
11: athrow
14: astore_1
19:  invokevirtual cfg.Config.getPort ()I (10)
22:  sipush 8000
25:  if_icmplt #38
28:  astore_1
29:  invokevirtual cfg.Config.getPort ()I (10)
32:  sipush 8005
35:  if_icmplt #48
38:  new java.lang.Exception (7)
41:  dup
42:  ldc "Invalid port." (11)
44:  invokevirtual java.lang.Exception (java.lang.String)
47:  athrow
50:  astore_0
51:  astore_1
52:  invokevirtual cfg.Controllercfg Lcfg/Config; (2)
55:  astore_0
54:  astore_1
55:  invokevirtual cfg.Config.getSignalCount ()I (6)
58:  newarray <int>
60:  putfield cfg.Controller.signals [I (3)
63:  return
```

Figure: Java Bytecode and corresponding Control-Flow Graph.
Methodology Overview

foreach class under test do
  instrument for structural tracing;
generate control-flow graphs;
identify test cluster;
generate EMCDGs and function sets;
foreach method under test do
  repeat
    reevaluate weight of CFG nodes;
generate individuals;
  foreach individual do
    generate method call sequence;
generate test case;
compose and execute test case;
trace CFG nodes hit;
evaluate test case;
remove hits from remaining nodes list;
recombine and mutate individuals;
until stopping criteria is met ;
Outline

Introduction
Software Testing
Evolutionary Algorithms
Evolutionary Testing

Approach and Methodology

Test Case Evaluation
Numerical Formulation of the Test Goal
Feasible and Unfeasible Test Cases
Weight Reevaluation

Experimental Studies
Probabilities of Operators
Evaluation Parameters
Discussion

Conclusions and Future Work
Numerical Formulation of the Test Goal

- Metaheuristic algorithms require a numerical formulation of the test goal – i.e., a **fitness function**.

Search Goal

- The **primary goal** is finding a set of test cases that achieves full structural coverage of the test object.
- The **quality of test cases** is related to the structural entities of the method under test which are the current targets of the evolutionary search.
Numerical Formulation of the Test Goal

- However, the execution of test cases may abort prematurely if a runtime exception is thrown during execution.
- When this happens, it is not possible to trace the structures traversed because the final instruction of the method call sequence is not reached.

Example

```java
Stack stack0 = new Stack();
String string1 = "HelloWorld!";
int int2 = stack0.search(string1);
Object object3 = stack0.pop();  \Rightarrow EmptyStackException
Object object4 = stack0.pop();
Object object5 = stack0.peek();
```
Feasible and Unfeasible Test Cases

- **Feasible Test Cases** are effectively executed and terminate with a call to the method under test.
- **Unfeasible Test Cases** abort prematurely because a runtime exception is thrown.
- Longer and more intricate test cases are more prone to throw runtime exceptions.
- However, complex method call sequences are often needed for transversing certain problem nodes.
- If unfeasible test cases are blindly penalised, the definition of elaborate state scenarios test cases will be disencouraged.
Feasible and Unfeasible Test Cases

Feasible Test Case Evaluation

\[ \text{Fitness}_{\text{feasible}}(t) = \frac{\sum_{h \in H_t} W_h}{|H_t|} \]

- i.e., fitness := the average weight of the cfg nodes traversed.

Unfeasible Test Case Evaluation

\[ \text{Fitness}_{\text{unfeasible}}(t) = \beta + \frac{(\text{seqLen}_t - \text{exInd}_t) \times 100}{\text{seqLen}_t} \]

- i.e., fitness := percentage of instructions executed plus the unfeasible penalty constant \( \beta \).
Weight Reevaluation

- The issue of steering the search towards the traversal of interesting CFG nodes and paths was addressed by assigning weights to the CFG nodes.
- The higher the weight of a given node the higher the cost of exercising it, and hence the higher the cost of traversing the corresponding control-flow path.
A Strategy for Evaluating Feasible and Unfeasible Test Cases for the Evolutionary Testing of Object-Oriented Software

Test Case Evaluation

Weight Reevaluation

Weight Reevaluation

\[ W_{ni} = (\alpha W_{ni}) \left( \frac{\text{hit}C_{ni}}{|T|} + 1 \right) \left( \sum_{x \in N_{ni}} \frac{W_x}{|N_{ni}| \times \frac{W_{init}}{2}} \right) \]

- i.e., at the beginning of each generation the weight of a given node is multiplied by:
  - the **weight decrease constant** value $\alpha$, so as to decrease the weight of all CFG nodes indiscriminately;
  - the **hit count factor**, which worsens the weight of recurrently hit CFG nodes;
  - the **path factor**, which improves the weight of nodes that lead to interesting nodes and belong to interesting paths.
Outline

Introduction
  Software Testing
  Evolutionary Algorithms
  Evolutionary Testing

Approach and Methodology

Test Case Evaluation
  Numerical Formulation of the Test Goal
  Feasible and Unfeasible Test Cases
  Weight Reevaluation

Experimental Studies
  Probabilities of Operators
  Evaluation Parameters
  Discussion

Conclusions and Future Work
Discussion

- Automatic test case generation using search-based techniques is a difficult subject.
- Finding a good balance between the intensification and the diversification of the search is the key to the definition of a good strategy.
- The main task of the evolutionary operators is that of diversifying the search, allowing it to browse through a wider area of the search landscape.
- The task of intensifying the search and promoting the transversal of unexercised CFG nodes is performed by assigning weights to CFG nodes.
Experimental Studies

- Experiments were performed on the Stack class of the java.util package of JDK 1.4.2.

- Its public API is composed by five public methods: boolean empty(), Object peek(), Object pop(), Object push(Object item) and int search(Object o).

- The main objectives were those of experimenting with different configurations for
  - the probabilities of evolutionary operators – mutation, reproduction and crossover;
  - the values of the test case evaluation parameters – the weight decrease constant $\alpha$ and the unfeasible penalty constant $\beta$. 
Probabilities of Operators

- Distinct parametrizations:
  - high probability of selecting the mutation pipeline;
  - high probability of selecting the crossover pipeline;
  - high probability of selecting the reproduction pipeline;
  - equal probabilities of selecting either pipeline.

<table>
<thead>
<tr>
<th>MUT</th>
<th>r:0.1 c:0.1 m:0.8</th>
<th>r:0.8 c:0.1 m:0.1</th>
<th>r:0.1 c:0.8 m:0.1</th>
<th>r:0.33 c:0.33 m:0.34</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%full</td>
<td>#gens</td>
<td>%full</td>
<td>#gens</td>
</tr>
<tr>
<td>empty</td>
<td>100%</td>
<td>10.2</td>
<td>100%</td>
<td>11.2</td>
</tr>
<tr>
<td>peek</td>
<td>100%</td>
<td>6.6</td>
<td>100%</td>
<td>10.7</td>
</tr>
<tr>
<td>pop</td>
<td>100%</td>
<td>6.5</td>
<td>100%</td>
<td>8.9</td>
</tr>
<tr>
<td>push</td>
<td>100%</td>
<td>20.6</td>
<td>57%</td>
<td>16.4</td>
</tr>
<tr>
<td>search</td>
<td>95%</td>
<td>48.9</td>
<td>57%</td>
<td>48.2</td>
</tr>
</tbody>
</table>

- The results show that the strategy of assigning balanced probabilities to the all of the breeding pipelines yields better results.
Evaluation Parameters

The following values were used:

- $\alpha$ – 0.1, 0.5, and 0.9;
- $\beta$ – 0, 150, and 300.

<table>
<thead>
<tr>
<th></th>
<th>a=0.1</th>
<th>b=0</th>
<th>a=0.5</th>
<th>a=0.9</th>
<th>b=0</th>
<th>a=0.5</th>
<th>a=0.9</th>
<th>b=300</th>
<th>a=0.5</th>
<th>a=0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>5.2</td>
<td>5.5</td>
<td>4.8</td>
<td></td>
<td>4.5</td>
<td>4.5</td>
<td>4.5</td>
<td></td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>peek</td>
<td>3.0</td>
<td>3.5</td>
<td>3.4</td>
<td></td>
<td>2.7</td>
<td>2.7</td>
<td>2.8</td>
<td></td>
<td>3.2</td>
<td>3.2</td>
</tr>
<tr>
<td>pop</td>
<td>2.8</td>
<td>3.2</td>
<td>3.1</td>
<td></td>
<td>2.4</td>
<td>2.4</td>
<td>2.8</td>
<td></td>
<td>3.1</td>
<td>3.1</td>
</tr>
<tr>
<td>push</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
<td></td>
<td>5.2</td>
<td>5.2</td>
<td>2.5</td>
<td></td>
<td>5.2</td>
<td>5.2</td>
</tr>
<tr>
<td>search</td>
<td>17.5</td>
<td>18.4</td>
<td>22.3</td>
<td></td>
<td>15.5</td>
<td>15.5</td>
<td>18.7</td>
<td></td>
<td>15.8</td>
<td>20.8</td>
</tr>
<tr>
<td>average</td>
<td>6.7</td>
<td>7.1</td>
<td>7.7</td>
<td></td>
<td>6.0</td>
<td>6.0</td>
<td>6.2</td>
<td></td>
<td>6.4</td>
<td>7.4</td>
</tr>
</tbody>
</table>

The results show that the best configuration for the test case evaluation parameters is that of assigning a low value to $\alpha$ (0.1 and 0.5 yielded the best results) and a value of approximately 150 to $\beta$. 
Discussion

Our strategy:

- causes the **fitness of feasible test cases** that **exercise recurrently traversed structures** to **fluctuate** throughout the search process.

- allows **unfeasible test cases** to be **considered** at certain points of the evolutionary search – once the **feasible test cases** being bred **cease to be interesting**.
A Strategy for Evaluating Feasible and Unfeasible Test Cases for the Evolutionary Testing of Object-Oriented Software

Conclusions and Future Work

Outline

Introduction
  Software Testing
  Evolutionary Algorithms
  Evolutionary Testing

Approach and Methodology

Test Case Evaluation
  Numerical Formulation of the Test Goal
  Feasible and Unfeasible Test Cases
  Weight Reevaluation

Experimental Studies
  Probabilities of Operators
  Evaluation Parameters
  Discussion

Conclusions and Future Work
Conclusions

- Search-Based Test Case Generation is an emerging methodology for automatically generating quality test data.

- However, the state problem of OO programs requires the definition of carefully fine-tuned methodologies → the transversal of problematic structures must be promoted.

- Complex method call sequences are often needed for traversing difficult control-flow paths.

- If unfeasible test cases are blindly penalised, the definition of elaborate state scenarios test cases will be disencouraged.
Conclusions

- We proposed tackling this problem by defining weighted CFG nodes, and dynamically reevaluating their weights every generation.

- The test case evaluation parameters $\alpha$ and $\beta$ and the evolutionary operators’ selection probabilities also play a central role in the test case generation process.

- Our strategy allows unfeasible test cases to be considered at certain points of the evolutionary search – once the feasible test cases being bred cease to be interesting.

- In conjunction with the impact of the evolutionary operators, a good compromise between the intensification and diversification of the search can be achieved.
Future Work

The most promising research-related challenges that lie ahead of us seem to be the following:

- **Input Domain Reduction** - deals with removing irrelevant variables from a given test data generation problem, thereby reducing the size of the search space.

- **Search Space Sampling** - deals with the inclusion of all the relevant variables to a given test object into test data generation problem, so as to enable the coverage of the entire search space.
For Further Reading I

B. Beizer.
*Software Testing Techniques.*

B. Eckel.
*Thinking in Java.*

Sofya: Supporting Rapid Development of Dynamic Program Analyses for Java.


D. J. Montana.
Strongly Typed Genetic Programming.

S. Wappler and J. Wegener.
Evolutionary Unit Testing Of Object-Oriented Software Using A Hybrid Evolutionary Algorithm.

S. Wappler and J. Wegener.
Evolutionary Unit Testing of Object-Oriented Software Using Strongly-Typed gGnetic Programming.
For Further Reading II

Using Dynamic Analysis of Java Bytecode for Evolutionary Object-Oriented Unit Testing.


An Evolutionary Approach for Performing Structural Unit-Testing on Third-Party Object-Oriented Java Software.


J. C. B. Ribeiro.
Search-Based Test Case Generation for Object-Oriented Java Software Using Strongly-Typed Genetic Programming.