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

José Ribeiro[1] Mário Zenha-Rela[2] Francisco Vega[3]

[1]Polytechnic Institute of Leiria, Portugal

[2]University of Coimbra, Portugal

[3]University of Extremadura, Spain

Automation of Software Test, 2008

ICSE LEIPZIG 2008

Outline

Outline

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,
  - Mutatation, 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.

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

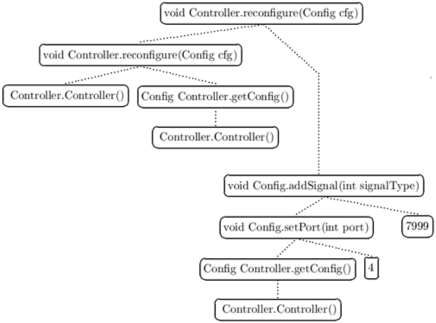
Our Approach

- **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

```
public void reconfigure(Config cfg)
0:    aload_1
1:    invokevirtual cfg.Config.getSignalCount ()I (6)
4:    iconst_5
5:    if_icmple #18
8:    new <java.lang.Exception> (7)
11:   dup
12:   ldc "Too many signals." (8)
14:   invokespecial java.lang.Exception (java.lang.String)
17:   athrow
18:   aload_1
19:   invokevirtual cfg.Config.getPort ()I (10)
22:   sipush 8000
25:   if_icmplt #38
28:   aload_1
29:   invokevirtual cfg.Config.getPort ()I (10)
32:   sipush 8005
35:   if_icmple #48
38:   new <java.lang.Exception> (7)
41:   dup
42:   ldc "Invalid port." (11)
44:   invokespecial java.lang.Exception (java.lang.String)
47:   athrow
48:   aload_0
49:   aload_1
50:   putfield cfg.Controller.cfg Lcfg/Config; (2)
53:   aload_0
54:   aload_1
55:   invokevirtual cfg.Config.getSignalCount ()I (6)
58:   newarray <int>
60:   putfield cfg.Controller.signals [I (3)
63:   return
```

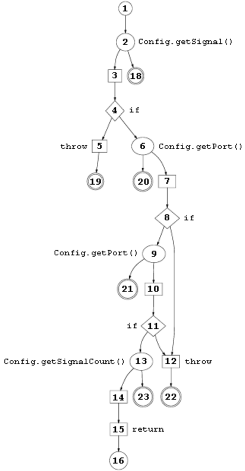


Figure: Java Bytecode and corresponding Control-Flow Graph.

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;
                compile 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 ;
```

ICSE LEIPZIG 2008

Outline

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

```
Stack stack0 = new Stack();
String string1 = "HelloWorld!";
int int2 = stack0.search(string1);
Object object3 = stack0.pop(); ⇒ 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.**

ICSE LEIPZIG 2008

Feasible and Unfeasible Test Cases

Feasible Test Case Evaluation

$$Fitness_{feasible}(t) = \frac{\sum_{h \in H_t} W_h}{|H_t|}$$

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

Unfeasible Test Case Evaluation

$$Fitness_{unfeasible}(t) = \beta + \frac{(seqLen_t - exInd_t) \times 100}{seqLen_t}$$

- i.e., fitness := percentage of instructions executed plus the *unfeasible penalty constant* β.

Weight Reevaluation

- The issue of steering the search towards the traversal of interesting CFG nodes and paths was address 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 transversing the corresponding control-flow path.

Weight Reevaluation

Weight Reevaluation

$$W_{ni} = (\alpha W_{ni}) \left(\frac{hitC_{ni}}{|T|} + 1 \right) \left(\frac{\sum_{x \in N_s^{ni}} W_x}{\left| N_s^{ni} \right| \times \frac{W_{init}}{2}} \right)$$

- i.e., at the beginning of each generation the weight of a given node is multiplied by:
  - the **weight decrease constant** value α, 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

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 `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* α and the *unfeasible penalty constant* β.

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.

	r:0.1 c:0.1 m:0.8		r:0.8 c:0.1 m:0.1		r:0.1 c:0.8 m:0.1		r:0.33 c:0.33 m:0.34	
MUT	**%full**	**#gens**	**%full**	**#gens**	**%full**	**#gens**	**%full**	**#gens**
`empty`	100%	10.2	100%	11.2	100%	17.5	100%	4.5
`peek`	100%	6.6	100%	10.7	100%	9.4	100%	2.8
`pop`	100%	6.5	100%	8.9	100%	8.6	100%	2.8
`push`	100%	20.6	57%	16.4	95%	37.2	100%	2.5
`search`	95%	48.9	57%	48.2	82%	98.8	100%	18.7

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

ICSE LEIPZIG 2008

Evaluation Parameters

- The following values were used:
  - α – 0.1, 0.5, and 0.9;
  - β – 0, 150, and 300.

	b=0			b=150			b=300		
	a=0.1	a=0.5	a=0.9	a=0.1	a=0.5	a=0.9	a=0.1	a=0.5	a=0.9
`empty`	5.2	5.5	4.8	4.5	4.5	4.5	5.0	5.0	4.9
`peek`	3.0	3.5	3.4	2.7	2.7	2.8	3.2	3.2	3.0
`pop`	2.8	3.2	3.1	2.4	2.4	2.8	3.1	3.1	2.5
`push`	5.2	5.2	5.2	5.2	5.2	2.5	5.2	5.2	5.2
`search`	17.5	18.4	22.3	15.5	15.5	18.7	15.8	20.8	22.1
average	6.7	7.1	7.7	6.0	6.0	6.2	6.4	7.4	7.5

- **The results show that the best configuration for the test case evaluation parameters is that of assigning a low value to α (0.1 and 0.5 yielded the best results) and a value of approximately 150 to β.**

ICSE LEIPZIG 2008

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**.

Outline

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 $\Longrightarrow$ 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 α and β 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.
John Wiley & Sons, Inc., New York, NY, USA, 1990.

B. Eckel.
Thinking in Java.
Prentice Hall Professional Technical Reference, 2002.

A. Kinneer, M. B. Dwyer, and G. Rothermel.
Sofya: Supporting Rapid Development of Dynamic Program Analyses for Java.
In *ICSE COMPANION '07: Companion to the Proceedings of the 29th International Conference on Software Engineering*, pages 51–52, Washington, DC, USA, 2007. IEEE Computer Society.

S. Luke.
ECJ 16: A Java Evolutionary Computation Library. http://cs.gmu.edu/∼eclab/projects/ecj/, 2007.

D. J. Montana.
Strongly Typed Genetic Programming.
Evolutionary Computation, 3(2):199–230, 1995.

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

S. Wappler and J. Wegener.
Evolutionary Unit Testing of Object-Oriented Software Using Strongly-Typed gGnetic Programming.
In *GECCO '06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 1925–1932, New York, NY, USA, 2006. ACM Press.

For Further Reading II

- J. C. B. Ribeiro, F. F. de Vega, and M. Z. Rela.
  Using Dynamic Analysis of Java Bytecode for Evolutionary Object-Oriented Unit Testing.
  In *SBRC WTF 2007: Proceedings of the 8th Workshop on Testing and Fault Tolerance at the 25th Brazilian Symposium on Computer Networks and Distributed Systems*, pages 143–156. Brazilian Computer Society (SBC), 2007.
- J. C. B. Ribeiro, M. Zenha-Rela, and F. F. de Vega.
  eCrash: A Framework for Performing Evolutionary Testing on Third-Party Java Components.
  In *JAEM'07: Proceedings of the I Jornadas sobre Algoritmos Evolutivos y Metaheuristicas at the II Congreso Español de Informática*, pages 137–144, 2007.
- J. C. B. Ribeiro, M. Zenha-Rela, and F. F. de Vega.
  An Evolutionary Approach for Performing Structural Unit-Testing on Third-Party Object-Oriented Java Software.
  In *NICSO 2007: Proceedings of the 2nd International Workshop on Nature Inspired Cooperative Strategies for Optimization (to appear)*, Studies in Computational Intelligence. Springer-Verlag, 11 2007.
- J. C. B. Ribeiro, M. Zenha-Rela, and F. F. de Vega.
  Strongly-Typed Genetic Programming and Purity Analysis: Input Domain Reduction for Evolutionary Testing Problems.
  In *GECCO'08: Proceedings of the Genetic and Evolutionary Computation Conference (to appear)*, 7 2008.
- J. C. B. Ribeiro.
  Search-Based Test Case Generation for Object-Oriented Java Software Using Strongly-Typed Genetic Programming.
  In *GECCO'08: Proceedings of the Genetic and Evolutionary Computation Conference (to appear)*, 7 2008.