An Evolutionary Approach For Performing Structural Unit-Testing On Third-Party Object-Oriented Java Software

José Carlos Ribeiro[1], Mário Zenha-Rela[2], and Francisco Fernández de Vega[3]

[1] Polytechnic Institute of Leiria (IPL)
Morro do Lena, Alto do Vieiro, Leiria, Portugal
`jose.ribeiro@estg.ipleiria.pt`
[2] University of Coimbra (UC)
CISUC, DEI, 3030-290, Coimbra, Portugal
`mzrela@dei.uc.pt`
[3] University of Extremadura (UNEX)
C/ Sta Teresa de Jornet, 38, Mérida, Spain
`fcofdez@unex.es`

Summary. Evolutionary Testing is an emerging methodology for automatically generating high quality test data. The focus of this paper is on presenting an approach for generating test cases for the unit-testing of object-oriented programs, with basis on the information provided by the structural analysis and interpretation of Java bytecode and on the dynamic execution of the instrumented test object. The rationale for working at the bytecode level is that even when the source code is unavailable, insight can still be obtained and used to guide the search-based test case generation process. Test cases are represented using the Strongly Typed Genetic Programming paradigm, which effectively mimics the polymorphic relationships, inheritance dependences and method argument constraints of object-oriented programs.

1 Introduction

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; automating this process is, therefore, vital to advance the state-of-the-art in software testing. In the particular case of unit-testing, individual application objects or methods are tested in an isolated environment; its goal is to warrant the robustness of the smallest units of the program under test. Distinct test approaches include functional (black-box) and structural (white-box) testing. Black-box testing is concerned with showing the conformity between the implementation and its functional specification; with *white-box testing* techniques, test case design is performed with basis on the program structure. When white-box testing is performed, the metrics for measuring the thoroughness of a given test set can be extracted from the structure of the target object's source code, or even from compiled code. Traditional white-box criteria include structural (e.g. statement, branch) coverage and data-flow coverage. The basic idea is to ensure that all of the control elements in a program are executed by a given test set, providing evidence of the quality of the testing activity.

The evaluation of test data suitability using structural criteria generally requires the definition of an underlying model for program representation – usually a control-flow graph (CFG). The observations needed to assemble the metrics required for the evaluation can be collected by abstracting and modeling the behaviours programs exhibit during execution, either by static or dynamic analysis techniques. Static analysis involves the construction and analysis of an abstract mathematical model of the system (e.g. symbolic execution); in contrast, *dynamic analysis* involves executing the actual test object and monitoring its behaviour. Dynamic monitoring of structural entities can be achieved by instrumenting the test object, and tracing the execution of the structural entities transversed during execution. Instrumentation is performed by inserting probes in the test object; in Java software, this operation can be effectively performed at the Java bytecode level.

Java bytecode is an assembly-like language that retains much of the high-level information about the original source program [1]. Class files (i.e. compiled Java programs containing bytecode information) are a portable binary representation that contains class related data, such as information about the variables and constants and the bytecode instructions of each method. Given that the target object's source code is often unavailable, working at the bytecode level allows broadening the scope of applicability of software testing tools; they can be used, for instance, to perform structural testing on third-party and COTS Java components. In addition, bytecode can be seen as an intermediate language, so the analysis performed at this level can be mapped back to the high-level language that generated the bytecode.

The focus of this work is precisely on the generation test data by employing evolutionary search techniques, with basis on the information provided by the structural analysis and interpretation of the Java bytecode and on the dynamic execution of the instrumented test object. The application of evolutionary algorithms to test data generation is often referred to as *evolutionary testing* [2, 3]. In evolutionary testing, meta-heuristic search techniques are employed to select or generate test data. The search space is the input domain of the test object, and the problem is to find a (minimal) set of test cases that satisfies a certain test criterion.

In the particular case of object-oriented programs, a sequence of method invocations is required to cover the test goal and the participating objects may have to be put into particular states in order for the test scenario to be processed in the desired way. The most pressing challenge faced by search-based test case generation is the *state problem* [4], which occurs with methods that exhibit state-like qualities by storing information in internal variables. Such variables are hidden from the optimization process, because they are protected from external manipulation using access modifiers (e.g. *getter* and *setter* methods). The only way to change their values is through execution of statements that perform assignments to them.

Evolutionary algorithms have been applied successfully to the search for quality test data in the field object-oriented unit-testing. Approaches have been proposed that focus on the usage of Genetic Algorithms [5], Ant Colony Optimization [6], Universal Evolutionary Algorithms [7], Genetic Programming [8], and on testing Container classes [9]. Of particular interest to our research is the work of Wappler *et. al* [10, 11], who proposed a methodology in which potential solutions are encoded using the Strongly Typed Genetic Programming (STGP) paradigm [12], with method call sequences being represented by STGP trees; these trees are able to express the call dependences of the methods that are relevant for a given test object. The STGP mechanism assures that only compilable programs are generated; to account for polymorphic relationships which exist due to inheritance relations, the STGP types used by the function set are specified in correspondence to the type hierarchy of the test cluster classes. The fitness function does need, however, to incorporate a penalty mechanism for test cases which include method call sequences that throw exceptions during the program execution – i.e. runtime exceptions.

2 Our approach for performing evolutionary structural unit-testing on third-party object-oriented software

This chapter presents the rationale and introduces our methodology for performing evolutionary structural unit-testing on third-party object-oriented software. Figure 1 summarizes the main phases of the testing process; the sub-chapters that follow describe the process in detail.

2.1 Static Analysis

Firstly, the test cluster's Java bytecode analysis is performed; it is at this step that the function set is defined, and hence it must precede the test set evolving and evaluation phases. The function set defines the restrictions that must be imposed to STGP nodes; specifically, they identify the children and return types of each node.

The first task is that of extracting the list of public methods from the test object's bytecode by means of the Java Reflection API; this list comprises the set of methods under test (MUTs) that are to be the subject of the unit-testing process. Secondly, the Extended Method Call Dependence Graph (EMCDG), which describes the method call dependences involved in the test

```
1. Static Analysis
 1.1. Test Cluster Analysis
 1.2. Test Object Analysis
 1.3. CFG Definition
 1.4. Test Object Instrumentation
2. foreach Generation
 2.1. CFG Nodes' Dynamic Weight Computation Phase
 2.2. Test Case Evolving Phase
  2.2.1. foreach Individual
   2.2.1.1. Test Case Generation
    2.2.1.1.1. Genetic Programming Tree Generation
    2.2.1.1.2. Genetic Programming Tree Linearization
    2.2.1.1.3. Test Case Generation
    2.2.1.1.4. Test Case Compilation
  2.2.1.2. Test Case Evaluation
    2.2.1.2.1. Test Case Execution
    2.2.1.2.2. Event Tracing
    2.2.1.2.3. Test Case Fitness Computation
```

Fig. 1. Methodology Overview.

case construction, is computed. Finally, the EMCDG is evaluated in order to define the function set.

For the definition of terminal nodes, the Ballista fault injection methodology [13] is employed. With the Ballista methodology, testing is performed by passing combinations of acceptable, boundary and exceptional inputs as parameters to the test object. The rationale for this inference is the perception that this constitutes a common programming pattern. This approach allows to effectively reduce the search space, which has been proved to improve results in many cases [14].

Control-flow graphs are used as the underlying model for program representation, and are built solely with basis on the information extracted from the Java bytecode of the test object. The CFG building procedure involves grouping bytecode instructions into a smaller set of Basic Instruction and Call CFG nodes, with the intention of simplifying the representation of the test object's control flow. Additionally, other types of CFG nodes, which represent virtual operations, are defined: Entry nodes, Exit nodes, and Return nodes. These virtual nodes encompass no bytecode instructions; they are used to represent certain control flow hypothesis. Instrumentation of the MUTs' bytecode for basic block analysis and structural event dispatch enables the observation of the CFG nodes transversed during a given program execution. Both the process of building the CFG and of instrumenting the MUT's are achieved with the aid of Sofya [15], a dynamic Java bytecode analysis framework.

2.2 Test Case Generation

For evolving the set of test cases, the ECJ package [16] is used. Test cases are evolved using the STGP paradigm, which effectively mimics the inheritance and polymorphic properties of object-oriented programs and enables the maintenance of call dependences when applying tree construction, mutation or crossover; the types specify which nodes can be used as a child of a node and which nodes can be exchanged between individuals.

Test cases are represented as GP trees; each GP individual contains a single GP tree. The first step involved in the generation of the test cases' source-code is the linearization of the GP trees using a depth-first transversal algorithm. The tree linearization process yields the ordered method call sequence; source-code generation is performed by translating this sequence into test cases using the information encoded into each node.

2.3 Test Case Evaluation

The evaluation of the quality of *feasible* test cases (i.e. those that do not throw runtime exceptions) is performed by comparing their trace information with the MUT's CFG. Event tracing is carried out by automatically executing the instrumented MUT using each generated test case as an "input"; relevant trace information includes the *Hit List* - i.e. the list of structural entities (CFG nodes) transversed. For *unfeasible* test cases, the fitness of the individual is calculated in terms of the distance between the runtime exception index (i.e. the position of the instruction that threw the exception) and the method call sequence length. Also, an *unfeasible penalty constant* is added to the final fitness value, in order to favour feasibility.

The algorithm for calculating the fitness of individuals is depicted in Figure 2. The CFG nodes *missing list* is initialized as being the complete CFG nodes list; when a particular CFG node is exercised by a test case, it is removed from the missing list. New test cases are generated as long as there are targets to be covered or a maximum number of generations is reached.

```
1. if test case is unfeasible
 1.1. compute method call distance (mcd)
  1.1.1. rti = get runtime exception index
  1.1.2. mcsl = get method call sequence length
  1.1.3. mcd = mcsl - rti
 1.2. fitness = (mcd * 100) / mcsl + UnfeasiblePenaltyConstant
2. else if test case is feasible
 2.1. totalWeight = 0
 2.2. foreach node in hitList
  2.2.1. totalWeight += weightOf(node)
  2.2.2. incrementHitCount(node)
 2.3. fitness = totalWeight / sizeOf(hitList)
 2.4. cfgNodesMissingList -= hitList
 2.5. if isEmpty(cfgNodesMissingList)
  2.5.1. found ideal individual
```

Fig. 2. Pseudo-code for the test case evaluation process.

The transversal of certain *problem nodes* requires the generation of complex test cases, which define elaborate state scenarios; alas, this often entails the generation of longer and more intricate method call sequences, which are more prone to throw runtime exceptions. Therefore, if unfeasible test cases are blindly penalised in favour of feasible ones the search landscape will be narrowed, thus hindering the possibility of transversing problem nodes. This issue 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 transversing the corresponding control-flow path.

The weights of every node are re-evaluated every generation in accordance to the algorithm depicted in Figure 3. With this approach, at the beginning of each generation the nodes' weight is firstly increased (worsened) to the direct proportion of the number of times that node was exercised by the individuals of the previous generation – with the intention of rising the cost of transversing frequently hit nodes; next, the nodes' weight is decreased in a *weight decrease constant* value – and consequently, nodes with a low hit count will be favoured; the nodes' final weight is calculated as the average of its own weight and that of its successors – so as to lower the cost of nodes that lead to less explored paths.

```
1. foreach node in cfg
 1.1. totalSucessorsWeight = 0
 1.2. weightOf(node) *= 1 + (hitCount(node) / sizeOf(population))
 1.3. weightOf(node) *= WeightDecreaseConstant
 1.4. foreach successorNode in successorNodesListOf(node)
  1.4.1. totalSucessorsWeight += weightOf(successorNode)
  1.4.2. incrementSucessorCount(node)
 1.5. weightOf(node) = (weightOf(node) + totalSucessorsWeight)
                     / (sizeOf(successorNodesListOf(node)) + 1)
2. normalizeNodeWeights(cfg)
```

Fig. 3. Pseudo-code for the CFG nodes weight computation.

The dynamic re-evaluation of the CFG nodes' weight presents the obvious advantage of steering the evolutionary search towards the transversal of less explored (or unexplored) nodes and paths; on the other hand, it worsens the fitness of test cases that exercise recurrently transversed CFG nodes. In fact – and depending on the value of the unfeasible penalty constant – unfeasible test cases may be selected for breeding at certain points of the evolutionary search, thus favouring diversity. This methodology intends to address a pitfall observed in preliminary experiments, which indicated that to strong a bias towards the generation of feasible test cases hinders the possibility of exercising problem CFG nodes, since the search gets stuck at a local maximum.

3 Experimental Study

In order to validate and clarify our approach, experiments were performed on the custom-made "Controller and Config" test cluster proposed in [11], using the `Controller.reconfigure(Config)` public method as the MUT.

The test cluster analysis phase yielded the function set described in [11]; the terminal set was defined in accordance to the Ballista methodology, and included 13 STGP nodes containing constant integer values: *Tn* = {*`Integer.MAXVALUE`, `Integer.MINVALUE`, 0, 4, 5, 6, 7999, 8000, 8001, 8004, 8005, 8006*}. We emulated the Ballista methodology by identifying the definition of constants in the test object's bytecode, depicted in Figure 4 *(left)*; namely, instructions at positions 4, 22 and 32 (`iconst_5`; `sipush 8000`; `sipush 8005`) push the constant integer values 5, 8000 and 8005 onto the top of the operand stack. These values were considered to be potential boundaries for numerical condition evaluation – hence their inclusion and that of their immediate neighbours (4, 6; 7999, 8001; 8004, 8006). The same heuristic was employed for including `Integer.MAXVALUE`, `Integer.MINVALUE` and 0 numerical values into *Tn*.

The CFG definition phase yielded the graph depicted in Figure 4 *(rigth)*. Attaining full structural coverage of the MUT required transversing all the Basic Instruction (4, 5, 8, 11, 12, 15) and Call (2, 6, 9, 13) CFG nodes.

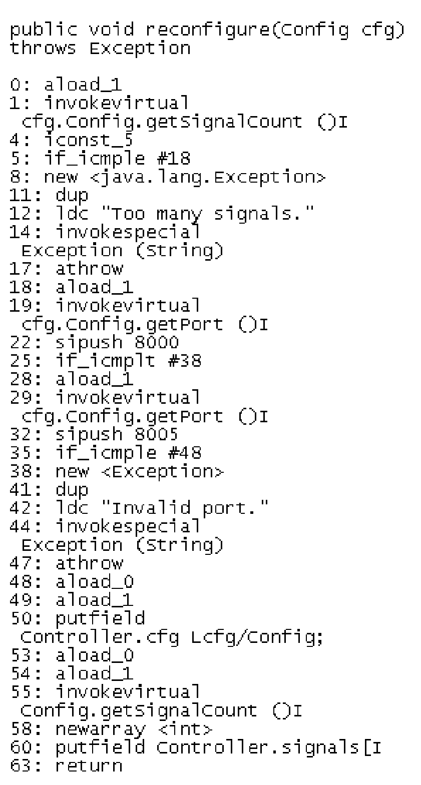
The evolutionary parameters for this experiment were defined as follows. The CFG nodes were initialized with a weight of 200; the weight decrease constant was set to 0.9, and the unfeasible penalty constant was defined as 100. ECJ was configured using a single population of 10 GP individuals. The breeding pipeline included strongly-typed versions of "Subtree Crossover" and "Point Mutation", and a simple reproduction operator; they were chosen with a probability of 0.6, 0.2 and 0.2 respectively. Tournament selection, with a size of 2.0, was employed as the selection method. The remaining configurations used were the Koza-style [17] parameters defined in ECJ by default. The search stopped if an ideal individual was found or after 200 generations.

Full structural coverage was achieved in all of the runs in an average of 27.6 generations (Table 1). The worst run found the ideal individual in 91 generations (seed 0), whilst in the best one all of the CFG nodes of the MUT were exercised in 4 generations (seeds 4 and 9).

Table 1. Number of generations required to find an ideal individual.

Seed	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	Average
normal	91	29	5	29	49	13	36	4	16	4	27.6
random	32	42	96	86	198	76	46	n/a	n/a	92	83.5

It could, however, be observed that 90% code coverage was achieved in an average of 2.3 generations; the remaining search process was spent trying to transverse problem CFG node 5. In fact, the CFG node 5 is paradigmatic of a problem node: its transversal accounts for only 10% of the fitness, and the branch that leads to it must be taken at Basic Instruction node 4 (subtype `if`); however, a test case requires 5 calls to the `Config.addSignal(int`



```
public void reconfigure(Config cfg)
throws Exception

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

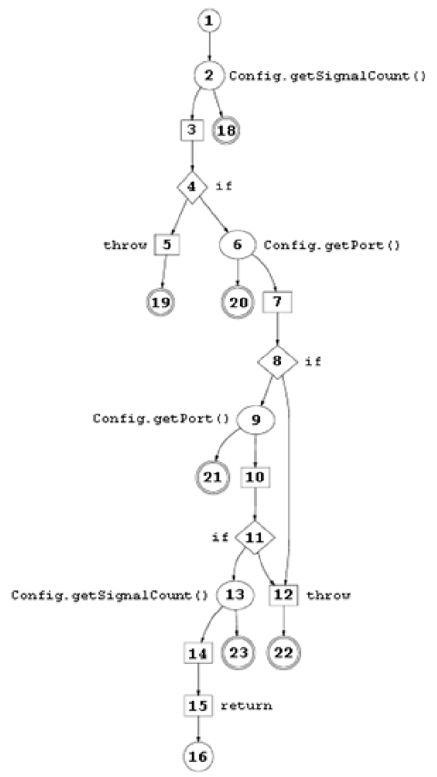


Fig. 4. Bytecode instructions *(left)* and CFG *(right)* for the `Controller.reconfigure(Config)` method of the "Controller and Config" test cluster.

`signal)` method of the `Config` object that will be used as a parameter in the MUT for this condition to be evaluated favourably.

Our methodology does, nevertheless, provide guidance towards the transversal of less explored paths and allows for unfeasible test cases to be produced at certain points of the evolutionary search, thus increasing diversity and promoting the definition of more complex scenarios. This phenomenon was particularly visible in the longest run, with seed 0 (Figure 5). In the initial generations, a high percentage of unfeasible test cases was produced; the search was then steered towards the generation of feasible test cases. 90% structural coverage was achieved in the 5th generation, with only CFG node 5 missing. Around generations 45-50, the weight of feasible test cases crossed the threshold defined by the unfeasible constant, thus allowing for unfeasible test cases to be selected for breeding.

The usefulness of the our methodology is particularly visible if the results are compared to those obtained using random search (Table 1). In order to

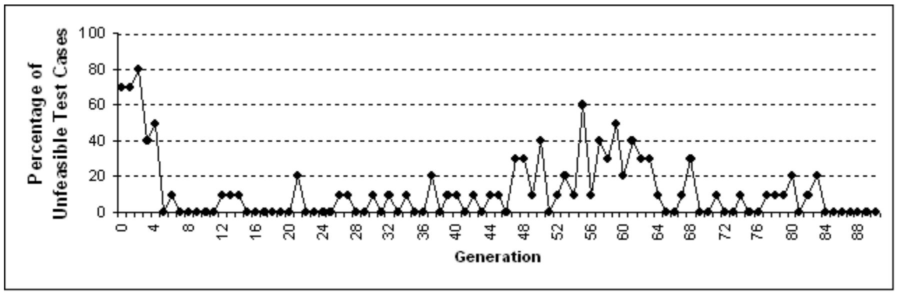


Fig. 5. Percentage of unfeasible test cases per generation for the longest running evolutionary search.

perform random search, the fitness was set to a constant value (in order to deprive the evolutionary search from guidance) with the remaining configurations and parameters being left unchanged. 10 runs were executed. Full structural coverage wasn't achieved in 20% of them; in the remaining, the average number of generations required to find an ideal individual was 83.5.

Finally, a battery of 10 runs was performed to validate the adequateness of using the Ballista methodology. In order to do so, the *Tn* terminal set was replaced a random integer value generator; the remaining configurations were left unaltered. In 6 of the 10 runs, 80% code coverage was achieved – CFG nodes 13 and 15 were never transversed; in the remaining 4 runs, the results yielded 70% code coverage – CFG nodes 5, 13 and 15 weren't exercised.

4 Conclusions and Future Work

This paper presents an evolutionary approach for the structural unit-testing of third-party object-oriented software. Relevant contributions include: the presentation of our methodology and underlying framework; the definition of a fitness function that effectively uses the insight obtained from the analysis of the test object's Java bytecode for search guidance; the proposal of methodologies for the dynamic re-evaluation the CFG nodes' weight; approaches for reducing the input domain of integer function parameter values. Experiments have been carried and quality solutions have been found, proving the pertinence of the approach and encouraging further studies.

Future work involves further research on the fitness function and domain reduction strategies, as well as on the minimization of the length of method call sequences so as to ease the user's task of defining assertions for the generated test cases, and on the identification and elimination of methods that do not alter the parameters' state from test cases' method call sequences.

References

1. Vincenzi, A.M.R., Delamaro, M.E., Maldonado, J.C., Wong, W.E.: Establishing structural testing criteria for java bytecode. Softw. Pract. Exper. **36**(14) (2006) 1513–1541
2. Mantere, T., Alander, J.T.: Evolutionary software engineering, a review. Appl. Soft Comput. **5**(3) (2005) 315–331
3. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability **14**(2) (2004) 105–156
4. McMinn, P., Holcombe, M.: The state problem for evolutionary testing (2003)
5. Tonella, P.: Evolutionary testing of classes. In: ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, ACM Press (2004) 119–128
6. Liu, X., Wang, B., Liu, H.: Evolutionary search in the context of object-oriented programs. In: MIC'05: Proceedings of the Sixth Metaheuristics International Conference. (2005)
7. Wappler, S., Lammermann, F.: Using evolutionary algorithms for the unit testing of object-oriented software. In: GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, New York, NY, USA, ACM Press (2005) 1053–1060
8. Seesing, A., Gro, H.G.: A genetic programming approach to automated test generation for object-oriented software. ITSSA **1**(2) (2006) 127–134
9. Arcuri, A., Yao, X.: Search based testing of containers for object-oriented software. Technical Report CSR-07-3 (2007)
10. Wappler, S., Wegener, J.: Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In: GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation, New York, NY, USA, ACM Press (2006) 1925–1932
11. Wappler, S., Wegener, J.: Evolutionary Unit Testing Of Object-Oriented Software Using A Hybrid Evolutionary Algorithm. In: CEC'06: Proceedings of the 2006 IEEE Congress on Evolutionary Computation, IEEE (2006) 851–858
12. Montana, D.J.: Strongly typed genetic programming. Technical Report #7866, 10 Moulton Street, Cambridge, MA 02138, USA (7 1993)
13. Kropp, N.P., Jr., P.J.K., Siewiorek, D.P.: Automated robustness testing of off-the-shelf software components. In: Symposium on Fault-Tolerant Computing. (1998) 230–239
14. Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Wegener, J.: The impact of input domain reduction on search-based test data generation. In: ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, New York, NY, USA, ACM Press (2007) 155–164
15. Kinneer, A., Dwyer, M., Rothermel, G.: Sofya: A flexible framework for development of dynamic program analysis for java software. Technical Report TR-UNL-CSE-2006-0006, University of Nebraska, Lincoln (4 2006)
16. Luke, S. ECJ 16: A Java evolutionary computation library. http://cs.gmu.edu/∼eclab/projects/ecj/ (2007)
17. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems). The MIT Press (December 1992)