eCrash: a Framework for Performing Evolutionary Testing on Third-Party Java Components

José Carlos Bregieiro Ribeiro
Polytechnic Institute of Leiria (IPL)
Morro do Lena, Alto do Vieiro
Leiria, Portugal
jose.ribeiro@estg.ipleiria.pt

Mário Zenha-Rela
University of Coimbra (UC)
CISUC, DEI, 3030-290
Coimbra, Portugal
mzrela@dei.uc.pt

Francisco Fernández de Vega
University of Extremadura (UNEX)
C/ Sta Teresa de Jornet, 38
Mérida, Spain
fcofdez@unex.es

Abstract

The focus of this paper is on presenting a tool for generating test data by employing evolutionary search techniques, with basis on the information provided by the structural analysis and interpretation of the Java bytecode of third-party Java components, and on the dynamic execution of the instrumented test object.

The main objective of this approach is that of evolving a set of test cases that yields full structural code coverage of the test object. Such a test set can be used for effectively performing the testing activity, providing confidence in the quality and robustness of the test object.

The rationale of working at the bytecode level is that even when the source code is unavailable structural testing requirements can still be derived, and used to assess the quality of a test set and to guide the evolutionary search towards reaching specific test goals.

1. Introduction

Software testing is an expensive process, typically consuming roughly half of the total costs involved in the software development process [1]. Locating suitable test data can be time consuming, difficult and expensive; automation of test data generation is, therefore, vital to advance the state-of-the-art in software testing.

Test data selection, generation and optimization deals with locating good test data for a particular test criterion. The assessment of the quality of a given set of test data can be achieved functionally (black-box testing) or structurally (white-box testing) [2].

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 [3]. 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; a test set that contains test cases that exercise all such elements is said to be adequate with respect to the corresponding criterion.

The evaluation of test data quality using white-box criteria generally requires the definition of an underlying model for program representation – usually a control-flow graph (CFG) [4]. The CFG is an abstract graph-based representation of a given method in a class – in the case of software testing, the test object. Evaluating the quality of a test set involves using CFGs to compute coverage metrics.

The observations needed to assemble the metrics required for the evaluation can be collected by abstracting and modelling the behaviours programs exhibit during execution [5], 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); testing is performed without executing the method under test, but rather this abstract model. This type of analysis is complex, and often incomplete due to the simplifications in the model. In contrast, dynamic analysis involves executing the actual test object and monitoring its behaviour; while it may not possible to draw general conclusions from dynamic analysis, it provides evidence of the successful operation of the software.

If dynamic analysis techniques are employed, the ability to observe program execution is

paramount. Events that need to be captured range from simple observations – such as execution of structural entities – to more complex examinations – such as thread and object creation, field manipulations, and object locking behaviour [4]. Dynamic monitoring of structural entities can be achieved by the instrumenting the test object, and dynamically tracing the execution of the structural entities transversed during execution.

Instrumentation in Java software is performed by inserting probes in the test object that log the entities exercised during execution. This operation can be performed either at the source-code level or 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 [6]. Class files (i.e. compiled Java programs containing bytecode information) are a portable binary representation that contains class related data, such as the class's name, its superclass's name, information about the variables and constants, and the bytecode instructions of each method.

Given that the target object's source code is often unavailable, performing instrumentation and CFG building with basis on bytecode allows broadening the scope of applicability of software testing tools. They can be used, for instance, to perform structural testing on third-party Java components. In addition, the bytecode can be seen as an intermediate language, so the analysis performed at this level can be mapped back to the original high-level language that generated the bytecode.

Evolutionary algorithms have been applied successfully to the search for quality test data in the field object-oriented unit-testing [7-11]. However, the application of search-based strategies in this area has not yet been investigated comprehensively; what's more, existing approaches work at the test object's source-code level. The evolutionary paradigm is expected to be equally suited if Java bytecode is employed as the basis for evolutionary search guidance and quality assessment.

The application of evolutionary algorithms to test data generation is often referred to as Evolutionary Testing [12, 13]. 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 sequence search space is an explosive space. Within the paradigm of object-orientation, the major concept is the object – which possesses attributes, constructors and methods. A test case for object-oriented software does not comprise only numerical test data; a sequence of constructor and method calls is also necessary.

Usually, multiple objects are involved in one single test case [11]:

- at least, an instance of the Class Under Test (CUT) is needed;
- additional objects, which are required (as parameters) for the instantiation of the CUT and for the invocation of the method under test (MUT), must be available, and for the creation of these additional objects more objects may be required;
- the participating objects may have to be put into particular states in order for the test scenario to be processed in the desired way and, consequently, method calls must be issued for these objects.

A fitness function for object-oriented evolutionary testing should evaluate test cases according to their ability to meet a given test goal. Fitness evaluation is, however hindered by the State Problem. The State Problem occurs with methods that exhibit state-like qualities by storing information in internal variables [14]; such variables are hidden from the optimization process, because they are protected from external manipulation using access modifiers (most notably "getter" and "setter" methods). The only way to change their values is through execution of statements that perform assignments to them.

In this paper, we present a prototypical tool – eCrash – that aims to provide a means to perform structural unit-testing on object-oriented software, using evolutionary techniques and with basis on the test object's bytecode. Firstly, in the following section, related work is reviewed. In section 3, the framework of our tool is outlined, and a case study that illustrates the methodology is described in chapter 4. The concluding section resumes the key ideas of this paper and presents some topics for future research.

2. Related Work

A first approach to the field of evolutionary testing of object-oriented software was presented in [10]; in this work, input sequences are generated using evolutionary algorithms for the white-box testing of classes. Genetic algorithms are the evolutionary approach employed, with potential solutions (test cases) being represented as chromosomes. A source-code representation is used, and an original evolutionary algorithm, with special evolutionary operators for recombination and mutation on a statement level (i.e. mutation operators insert or remove methods from a test program), is defined. A population of individuals, representing the test cases, is evolved in order to increase a measure of fitness, accounting for the ability of the test cases to satisfy a coverage criterion of choice. New test cases are generated as long as there are targets to be covered or a maximum execution time were reached.

However, the encapsulation problem was not addressed, and this proposal only dealt with a simple state problem. Additionally, with this approach, Universal Evolutionary Algorithms (i.e. evolutionary algorithms, provided by popular toolboxes, which are independent from the application domain and offer a variety of predefined, probabilistically well-proven evolutionary operators) could not be applied due to the usage of custom-made operators and original evolutionary algorithms.

An approach which employed an Ant Colony Optimization algorithm was presented in [9]. The focus is on the generation of the shortest method call sequence for a given test goal, under the constraint of state dependent behaviour and without violating encapsulation. Ant PathFinder, hybridizing Ant Colony Optimization and Multiagent Genetic Algorithms are employed. To cover those branches enclosed in private/protected methods without violating encapsulation, call chain analysis on class call graphs was introduced.

In [11] the focus was on the usage of Universal Evolutionary Algorithms. An encoding is proposed that represents object-oriented test cases as basic type value structures, allowing for the application of various search-based optimization techniques such as Hill Climbing or Simulated Annealing. The generated test cases can be transformed into test classes according to popular testing frameworks. Still, the suggested encoding did not prevent the generation of individuals which could not be decoded into test programs without errors; the fitness function used different penalty mechanisms in order to penalize invalid sequences and to guide the search towards regions that contained valid sequences. Due to the generation of infeasible sequences, the approach lacked efficiency for more complicated cases.

In [7] an approach in which potential solutions were encoded using a Strongly-Typed Genetic Programming (STGP) methodology was presented, with method call sequences being represented by method call trees; these trees are able to express the call dependences of the methods that are relevant for a given test object. 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 emphasis of this work is on sequence feasibility; the usage of STGP preserves feasibility throughout the entire search process. The fitness function does need, however, to incorporate a penalty mechanism for test cases which include method call sequences that generate runtime exceptions. The issue of runtime exceptions was precisely the main topic in [8].

The methodology proposed in [7, 8] yielded very encouraging results. For a simple custom-tailored test cluster, the set of generated test cases achieved 100% branch coverage; in a more complex scenario, four classes where tested and full coverage was achieved for all of the test objects.

In all of the abovementioned approaches, the underlying model for program representation (i.e. CFG) is built with basis on the test object's source-code; moreover, instrumentation of the test object for extracting tracing information is also performed at the source-code level. To the best of our knowledge, there are no evolutionary approaches to the unit-testing of object-oriented software that employ dynamic bytecode analysis to derive structural testing criteria.

The application of evolutionary algorithms and bytecode analysis for test automation was, nonetheless, studied in different scenarios. A black-box approach using program specifications written in JML was employed in [15], and [16] describes a methodology based on static analysis techniques.

3. Framework Overview

This focus of this paper is on presenting the framework of a tool (which we named "eCrash") for evolving test sets for structural unit-testing of third-party object-oriented software.

The ideas that lead to this approach were greatly inspired by the previous works of [6-8, 11, 17]. Test cases are evolved using a STGP mechanism, with the metrics required to evaluate their quality being collected at the bytecode level. The framework of our tool is outlined in Figure 1.

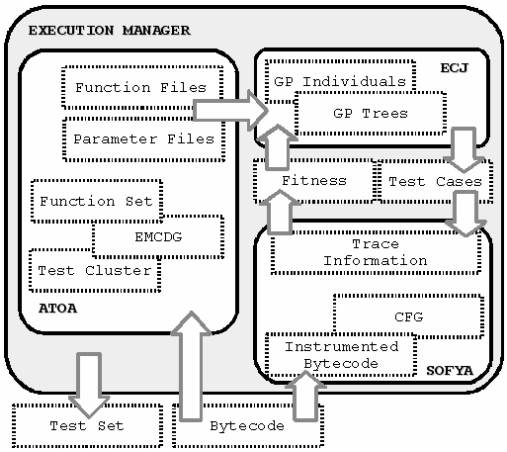


Figure 1. Framework overview

For evolving the set of test cases, the Evolutionary Computation in Java (ECJ) package [18] is used. ECJ is a research package that incorporates several Universal Evolutionary Algorithms, and includes built-in support for Set-Based STGP. It is highly flexible, having nearly all classes and their settings being dynamically determined at runtime by user provided Parameter files and Function Set files.

The process of CFG building, bytecode instrumentation and event tracing is achieved with the aid of Sofya [4], a dynamic Java bytecode analysis framework. The Sofya package provides implementations and tools for the construction of various kinds of graphs – most notably CFGs – and native capabilities for dispatching event streams of specified program observations, which include instrumentators, event dispatchers, and event selection filters for semantic and structural event streams. Additionally, it contains tools to perform various analyses using the outputs generated by its components (statistics, coverage reports, …) and to visualize the trace information produced by the executions of instrumented programs.

The test cluster analysis phase is performed by the "Automatic Test Object Analyser" (ATOA) module of the eCrash tool. It's main task is that of generating Parameter Files containing the constraints needed for the STGP system.

4. Case Study

In this experiment, the simple test cluster defined in [8] is used for demonstration purposes. The Controller.reconfigure(Config) public method was used as the method under test (MUT); its source code is depicted in Figure 2.

```
public void reconfigure(Config cfg) throws Exception {
  if( cfg.getSignalCount() > MAX_SIGNALS )
     throw new Exception("Too many signals.");
  if(cfg.getPort()<MIN_PORT||cfg.getPort()>MAX_PORT)
     throw new Exception("Invalid port.");
  this.cfg = cfg;
  signals = new int[cfg.getSignalCount()];
}
```

Figure 2. Method Under Test's source code [8]

4.1. Test Cluster Analysis

The test cluster's Java bytecode analysis is performed by the ATOA module of the eCrash framework; it is at this step that the Function and Terminal sets are defined, and hence it must precede the test set evolving and evaluation phases.

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 MUTs that are to be the subject of the unit-testing process. Secondly, the Extended Method Call Dependence Graph (EMCDG) is determined; this structure describes the method call dependences involved in the test case construction [7].

Function and Terminal sets are then computed for each of the MUTs by evaluating the EMCDG. These sets define the restrictions that must be imposed to STGP tree nodes; specifically, they identify the children and return types of each

node. This information is used to generate ECJ Parameter files that contain the constraints of the STGP system, and assures that the test cases' call dependences are taken into account.

```
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 3. Method Under Test's bytecode instructions

For this case study's MUT, the EMCDG analysis yielded the Function Set depicted in [7], which includes both the terminal and non-terminal STGP nodes involved in the method call sequence construction. A distinct approach was, however, employed for the definition of terminal nodes representing numerical values – the Ballista fault injection methodology [2].

With the Ballista methodology, testing is performed by passing combinations of acceptable, boundary and exceptional inputs as parameters to the test object via an ordinary method call.

With this in mind, 9 additional terminal nodes were defined for this MUT, containing the following constant values: 4, 5, 6; 7999, 8000, 8001; 8004, 8005, 8006. The analysis that lead to the definition of this sub-set of terminal nodes follows.

Bytecode instructions (Figure 3) at positions 4, 22 and 32 (iconst_5; sipush 8000; sipush 8005) push the integer values 5, 8000 and 8005 onto the top of the operand stack, for usage in posterior instructions of type "if". These constant values are, therefore, potential boundaries for numerical condition evaluation; the rationale for this inference is the perception that this constitutes a common programming pattern. This approach allows us to emulate the behaviour proposed by Ballista, as it is a step towards the definition of valid, invalid and boundary test cases – if integers 5, 8000 and 8005 are indeed boundaries in decision structures.

4.2. Test Set Representation and Generation

Test cases are represented as GP trees; test sets correspond to GP individuals, each containing a pre-defined number of GP trees. Individuals and trees are generated automatically by the ECJ tool, in conformity with the constraints imposed in the Parameter files.

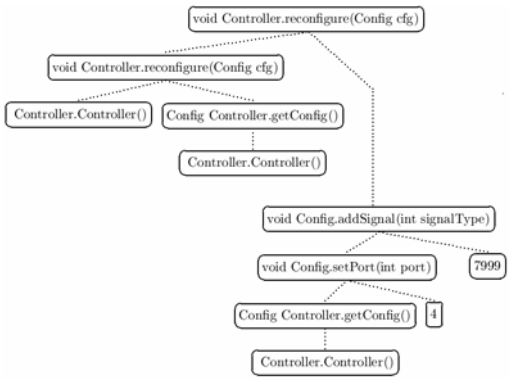


Figure 4. Example GP tree

The task of defining the number of GP trees (test cases) involves identifying all the problem blocks in the CFG – i.e. nodes at which execution takes a critical branch, making it impossible to reach a certain target node once the control flow has diverged. The minimum number of test cases is equal to the number of distinct control flow paths.

For the abovementioned MUT, the set of problem blocks includes blocks 4, 8 and 11 of the

CFG depicted in Figure 6 (Basic Instruction blocks of subtype "if") and hence the number of GP trees was set as 3 per GP individual. CFG definition and interpretation will be described in further detail in the next subchapter.

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 the method call sequence into test cases using the information encoded into each node. The STGP mechanism assures that only valid GP trees – i.e. that can be transformed into compilable test cases – were generated.

Figure 4 contain the an example GP tree generated by ECJ for this case study's MUT, and Figure 5 depicts the corresponding test case's source-code.

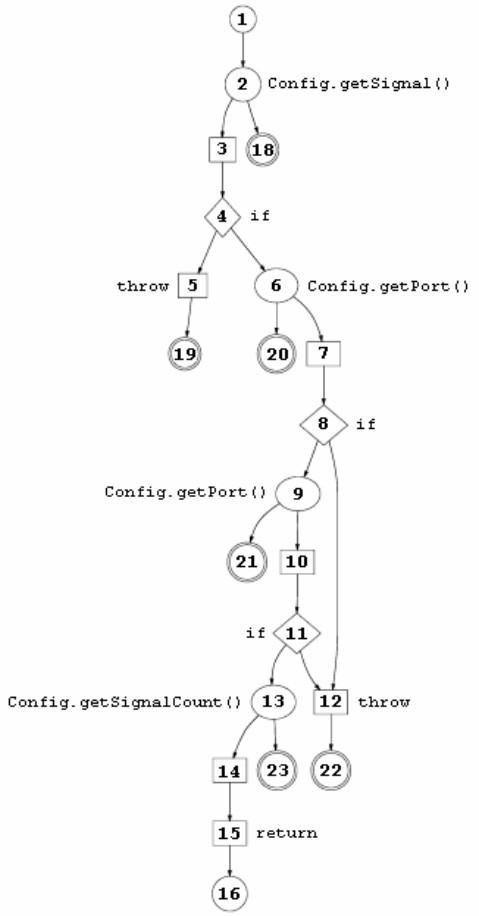
```
package testCases;
import testObject.*;

public class MainG0I2T2 {
	public static void main(String[] args) {
		try {
			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);
		} catch (Exception e) {
			System.err.println("MainG0I2T2: " + e);
}	}	}
```

Figure 5. Example test case

4.3. Test Set Evaluation and Fitness Definition

The main objective of this case study was that of conducting a successful evolutionary search for a test set that achieved full structural coverage – i.e. a test set that yields the transversal of all the Java bytecode instructions of the MUT.

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 evaluation of the quality of a given



Initial bc inst	final bc inst	node type	node subtype	node number
0	1	Call		2
4	5	Basic	If	4
8	17	Basic	Throw	5
18	19	Call		6
22	25	Basic	If	8
28	29	Call		9
32	35	Basic	If	11
38	47	Basic	Throw	12
48	55	Call		13
58	63	Basic	Return	15

Figure 6. Method Under Test's Control-Flow Graph; mapping between bytecode instructions, basic instruction blocks, basic instruction block subtypes, and node numbers in MUT's CFG

test set is, therefore, performed by comparing the trace information collected by the dynamic execution of the MUT against its CFG, with the purpose of verifying the coverage thoroughness achieved by that test set. The tasks of building the CFGs and of instrumenting the MUT's bytecode for basic block tracing and structural event dispatch both precede that of evolving test sets, and are performed with the aid of the Sofya tool.

The CFG building procedure involves grouping bytecode instructions into a smaller set of Basic Instruction and Call blocks, with the intention of easing the representation of the test object's control flow. Additionally, other types of blocks which represent virtual operations are defined: Entry blocks, Exit blocks, and Return blocks. These Virtual blocks encompass no bytecode instructions; they are used to represent certain control flow hypothesis. For this case study's MUT, Basic Instruction blocks (4, 5, 8, 11, 12, 15) and all the Call blocks (2, 6, 9, 13) of the CFG depicted in Figure 6 must be transversed in order to attain full structural coverage.

Instrumentation of the MUT's classes for basic block analysis and structural event dispatch enables the observation of the blocks transversed during a given program execution; event tracing is then performed by automatically executing the instrumented MUT using each generated test case as an "input", with the intention of collecting trace information with which to derive coverage metrics. Relevant trace information includes the list of blocks transversed (Hit List) in the MUT's CFG by the execution of each individual test case.

In our current approach, the Hit List is computed individually for each test case; the GP individual's overall fitness is calculated as the percentage of bytecode instructions exercised by the whole test set – i.e. the percentage of blocks transversed by the execution of all the test cases in the test set.

4.4. Experimental Observations

In this experiment, ECJ was configured using a single subpopulation of 5 GP individuals, with each individual containing 3 GP trees; each run stopped if an ideal individual was found or after 300 generations. The remaining parameters used were the Koza-style [19] definitions used in ECJ by default: Tournament Selection for Reproduction, One-Point Mutation and Sub-Tree Crossover, and Half/Full Tree Initialization.

The best run successfully achieved full structural coverage with 11 generations. The definition of Ballista-based terminal nodes proved to be valuable; in control runs, numerical values were generated randomly, and only 80% code coverage was achieved after 300 generations. For comparison purposes, ECJ was also parameterized using random mutation, reproduction, and crossover operators. 100% structural coverage was also achieved; however, the minimum number of generations required to do so was 78.

Still, some problems persist. In this experiment, it was possible to observe that if full structural coverage is not achieved in the initial generations, it's unlikely that it is achieved in that run – i.e. as generations evolve, the evolutionary search is steered towards a local maximum that hinders the possibility of achieving 100% code coverage.

This behaviour can be explained by the State Problem; the CFG's problem block 5 is paradigmatic. The transversal of this block accounts only for 10% of the fitness, and the branch that leads to it must be taken at Basic Instruction block 4 (sub-type "if"); however, a test case requires 5 calls to the `Config.addSignal(int signal)` method of the `Config` object that will be used as a parameter in the MUT for this condition to be evaluated favourably. The fitness function currently employed provides no guidance for this particular class of problems.

5. Conclusions and Future Work

This paper presents an evolutionary approach for the structural unit-testing of third-party object-oriented software. Preliminary experiments have been carried out and quality solutions have been found, proving the pertinence of the approach.

Future work involves addressing the State Problem, by implementing adequate fitness functions that can steer the evolutionary search towards individual test goals on the test object. This can be achieved by the definition of distance-based metrics [17], which can express how close the execution of a test case over the test object is to reaching a given test goal.

Further research must also be made on the topics of easing the user's task of defining assertions for the generated test cases (e.g. by minimizing the length of method call sequences), and on the usage of a set-typing mechanism for mimicking the polymorphic relations that exist amongst the test cluster's classes.

References

[1] Li, K. and M. Wu, Effective software test automation: developing an automated software testing tool. 2004, San Francisco, California ; London: Sybex. xx, 408 p.

[2] Kropp, N.P., P.J. Koopman, and D.P. Siewiorek. Automated Robustness Testing of Off the Shelf Software Components. in FTCS 98, IEEE. 1998.

[3] Vincenzi, A.M.R., et al., Coverage testing of Java programs and components. Special issue on new software composition concepts, 2005. 56(1-2): p. 211-230.

[4] Kinneer, A., M. Dwyer, and G. Rothermel, Sofya: A Flexible Framework for Development of Dynamic Program Analyses for Java Software. 2006, Department of Computer Science and Engineering, University of Nebraska - Lincoln.

[5] Tracey, N., et al., A search-based automated test-data generation framework for safety-critical systems. Systems engineering for business process change: new directions. 2002: Springer-Verlag New York, Inc.

[6] Vincenzi, A.M.R., et al., Establishing structural testing criteria for Java bytecode. Software Practice and Experience, 2006. 36(14): p. 1513-1541.

[7] Wappler, S. and J. Wegener, 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. 2006, ACM Press: Seattle, Washington, USA. p. 1925-1932.

[8] Wappler, S. and J. Wegener, Evolutionary Unit Testing Of Object-Oriented Software Using A Hybrid Evolutionary Algorithm, in Proceedings of the 2006 IEEE Congress on Evolutionary Computation. 2006, IEEE Press: Vancouver. p. 3193-3200.

[9] Liu, X., B. Wang, and H. Liu. Evolutionary search in the context of object-oriented programs. in MIC2005: The Sixth Metaheuristics International Conference. 2005. Vienna, Austria.

[10] Tonella, P., Evolutionary testing of classes, in ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. 2004, ACM Press: Boston, Massachusetts, USA. p. 119-128.

[11] Wappler, S. and F. Lammermann, Using evolutionary algorithms for the unit testing of object-oriented software, in GECCO '05: Proceedings of the 2005 conference on genetic and evolutionary computation. 2005, ACM Press. p. 1053-1060.

[12] Mantere, T. and J.T. Alander, Evolutionary software engineering, a review. Applied Soft Computing, 2005. 5(3): p. 315-331.

[13] McMinn, P., Search-based software test data generation: a survey. Software Testing, Verification and Reliability, 2004. 14(2): p. 105-156.

[14] McMinn, P. and M. Holcombe. The state problem for evolutionary testing. in Genetic and Evolutionary Computation Conference. 2003. Chicago, USA: Springer-Verlag.

[15] Cheon, Y., M.Y. Kim, and A. Perumandla, A complete automation of unit testing for Java programs, in Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05). 2005, CSREA Press: Las Vegas, Nevada, USA. p. 290-295.

[16] Muller, R.A., C. Lembeck, and H. Kuchen, A symbolic Java virtual machine for test case generation, in Proceedings of IASTED Conference on Software Engineering. 2004. p. 365-371.

[17] Wegener, J., A. Baresel, and H. Sthamer, Evolutionary test environment for automatic structural testing. Information & Software Technology, 2001. 43(14): p. 841-854.

[18] Luke, S., et al. ECJ 16: A Java evolutionary computation library. 2007 [cited; Available from: http://www.cs.gmu.edu/~eclab/projects/ecj/.

[19] Koza, J.R., Genetic programming : on the programming of computers by means of natural selection. Complex adaptive systems. 1992, Cambridge, Mass. ; London: MIT. xiv,819p.