Diploma de Estudios Avanzados

Evolutionary Unit-Testing Of Third-Party Object-Oriented Java Software

José Carlos Bregieiro Ribeiro

Universidad de Extremadura
España 2007

Organization:

Dept. de Tecnologías, Computadores y Comunicaciones; Escuela Politécnica; Universidad de Extremadura

Title:

Evolutionary Unit-Testing Of Third-Party Object-Oriented Java Software

Author:

José Carlos Bregieiro Ribeiro

Supervising Teacher:

Francisco Fernandéz de Vega

Line of Investigation:

Techniques for Planning the Improvement of the Efficiency of Evolutionary Algorithms (*Técnicas de Planificación para la Mejora de la Eficiencia de Algoritmos Evolutivos*)

Period:

2005/2007

Abstract

Evolutionary Testing is an emerging methodology for automatically generating high quality test data. The focus of this work is on presenting a search-based approach for the the unit-testing of third-party object-oriented Java software.

Test cases are represented and evolved using the Strongly Typed Genetic Programming 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.

Our strategy for evaluating the quality of test cases includes instrumenting the test object for basic block analysis and structural event dispatch, and executing the instrumented test object using the generated test cases as "inputs" – in order to collect trace information and derive coverage metrics.

Static analysis, instrumentation and execution tracing is performed solely with basis on the high-level information extracted from the Java Bytecode of the test object. Given that the test object's source code is often unavailable, working at the Bytecode level allows broadening the scope of applicability of our approach; it can be used, for instance, to perform structural testing on third-party Java components.

Test objects are represented internally by weighted control-flow graphs; strategies are introduced for favouring test cases that exercise problematic structures and difficult control-flow paths, which involve dynamic weight reevaluation. The aim is that of efficiently guiding the search process towards achieving full structural coverage – which often involves promoting the definition of complex and intricate test cases that define elaborate state scenarios.

The work performed so far allowed us to develop the prototype of a test case generation tool, called *eCrash*. Experiments have been carried and quality solutions have been found, proving the pertinence of our methodology and encouraging further studies.

List of Publications

Publications:

- José Carlos Bregieiro Ribeiro, Francisco Fernandéz de Vega, and Mário Zenha-Rela. "Using Dynamic Analysis of Java Bytecode for Evolutionary Object-Oriented Unit Testing", in Proceedings of the 8th Workshop on Testing and Fault Tolerance of the 25th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC WTF 2007), pages 143-156. ISBN:85-766-0119-1. Belém, Brazil, May 2007.

- José Carlos Bregieiro Ribeiro, Mário Zenha-Rela, and Francisco Fernandéz de Vega. "eCrash: a Framework for Performing Evolutionary Testing on Third-Party Java Components", in Proceedings of the I Jornadas sobre Algoritmos Evolutivos y Metaheuristicas of the II Congreso Español de Informática (CEDI JAEM 2007), pages 143-156. ISBN:978-84-9732-593-6. Zaragoza, Spain, September 2007.

- José Carlos Bregieiro Ribeiro, Mário Zenha-Rela, and Francisco Fernandez de Vega. "An Evolutionary Approach For Performing Structural Unit-Testing On Third-Party Object-Oriented Java Software", in Proceedings of International Workshop on Nature Inspired Cooperative Strategies for Optimization (NICSO 2007). Studies in Computational Intelligence Book Series, Springer-Verlag (*to appear*). Acireale, Italy, November 2007.

Related Work:

- José Carlos Bregieiro Ribeiro, Bruno Miguel Luís, and Mário Zenha-Rela. "Error propagation monitoring on windows mobile-based devices", in Proceedings of the Third Latin-American Symposium on Dependable Computing (LADC 2007), volume 4746/2007 of Lecture Notes in Computer Science, pages 111-122. ISBN:978-3-540-75293-6. Morelia, Mexico, September 2007.

Acknowledgements

I would like to start by thanking Professor Francisco Vega for giving me the privilege of working with him and for guiding me half-way towards the goal.

I also express my deepest gratitude to Professor Mário Zenha-Rela. I owe much of my research to his availability, his support, his enthusiasm.

I take opportunity for thanking mum, dad, and my (not so) little brother. I wish I could express my gratitude in words, but I wouldn't know where to begin.

Finally, I would like to send a big kiss to my girlfriend Marta, who has put up with more than her fair share throughout these last two (or should I say ten...) years.

Contents

1 Introduction **1**

2 Background **3**

2.1 Software Testing 3

2.1.1 Structural and Functional Testing 3

2.1.2 Static Analysis and Dynamic Analysis 4

2.1.3 Levels of Testing 5

2.1.4 Unit Testing 6

2.1.5 Object-Oriented Unit Testing 6

2.2 Evolutionary Algorithms 7

2.2.1 Genetic Algorithms 8

2.2.2 Genetic Programming 9

2.2.3 Strongly Typed Genetic Programming 9

2.3 Evolutionary Testing 10

2.3.1 The State Problem 10

2.4 Java Bytecode 11

3 Related Work **13**

4 Methodology and Framework **17**

4.1 Methodology 18

4.1.1 Static Analysis and Intrumentation 18

4.1.2 Test Case Generation 20

4.1.3 Test Case Evaluation 20

Weight Reevaluation 23

Evaluation of Feasible Test Cases 24

Evaluation of Unfeasible Test Cases 24

4.2 Framework Overview 24

5 Experimental Studies **27**

5.1 Case Study: Controller & Config 28

5.1.1 Setup 30
5.1.2 Results 32
5.2 Case Study: Stack 34
5.2.1 Setup 35
5.2.2 Results 35
Probabilities of Operators 35
Evaluation Parameters 37
5.3 Discussion 38

6 Future Work **41**
6.1 Method Call Sequence Separation 42
6.2 Input Domain Reduction 42
6.3 Search Domain Sampling 44

7 Conclusions **47**

A Teaching and Investigation Periods **55**
A.1 Design and Implementation of Reconfigurable Systems and Parallel Architectures 55
A.2 Grid Computing and Evolutionary Algorithms 57
A.3 Robotics and Artificial Vision 57

B Planning **59**

C Publications **63**
C.1 Using Dynamic Analysis of Java Bytecode for Evolutionary Object-Oriented Unit Testing (SBRC WTF 2007) 65
C.2 eCrash: a Framework for Performing Evolutionary Testing on Third-Party Java Components (CEDI JAEM 2007) 81
C.3 An Evolutionary Approach For Performing Structural Unit-Testing On Third-Party Object-Oriented Java Software (NICSO 2007) 91

D Certification Document – Grades Obtained During the Teaching and Investigation Periods **103**

List of Figures

4.1 Methodology overview. 19
4.2 Genotype example – Strongly Typed Genetic Programming tree. 21
4.3 Phenotype example – Method Call Sequence. 21
4.4 Example test case. 22
4.5 Framework overview. 25

5.1 Source code of the "Controller & Config" test object. 28
5.2 Bytecode of the `reconfigure` method. 30
5.3 Control-Flow Graph of the `reconfigure` method. 31
5.4 Percentage of unfeasible test cases per generation. 33
5.5 Results for the "Evaluation Parameters" experiment. 37

List of Tables

5.1 Function Set for the “Controller & Config” case study. 29
5.2 Mapping table between the Bytecode instructions and the Control-Flow Graph nodes for the `reconfigure` method. . . . 31
5.3 Number of generations required to attain full coverage. 33
5.4 Function Set for the `Stack` case study. 35
5.5 Statistics for the “Probabilites of Operators” experiment. . . . 36

6.1 Improved Function Set for the `Stack` case study. 44

A.1 Teaching Period. 56
A.2 Investigation Period. 56

B.1 High-level goals for the development of the thesis. 59
B.2 Low-level tasks for the development of the thesis. 61

List of Abbreviations

ATOA	Automatic Test Object Analyzer
CFG	Control Flow Graph
COTS	Commercial Off The Shelf
CUT	Class Under Test
ECJ	Evolutionary Computation in Java
EMCDG	Extended Method Call Dependence Graph
GP	Genetic Programming
GUI	Graphical User Interface
JML	Java Modeling Language
JVM	Java Virtual Machine
LIFO	Last In First Out
MCS	Method Call Sequence
MUT	Method Under Test
SAIM	Static Analysis and Instrumentation Module
SBTCG	Search-Based Test Case Generation
STGP	Strongly Typed Genetic Programming
TCEM	Test Case Evaluation Module
TCGM	Test Case Generation Module

Chapter 1

Introduction

Software testing is an expensive process, typically consuming roughly half of the total costs involved in the software development process – while adding nothing to the raw functionality of the final product. Yet, it remains the primary method through which confidence in software is achieved.

Test data selection and generation deals with locating *good* test data for a particular test criterion [38]. In industry, this process is often performed manually – with the responsibility of assessing the quality of a given software product falling on the software tester. However, 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.

The application of evolutionary algorithms to test data selection and generation is often referred to as "Evolutionary Testing" [23, 21] or "Search-Based Test Case Generation" [5, 10]. The search space is the input domain of the program under test, and the problem is to find a set of input data – called test cases – that satisfies a certain test criterion [6].

The focus of our on-going work [29, 31, 32] is precisely on presenting a search-based approach for automatically generating test cases for the unit-testing of object-oriented programs. This report aims to provide an overview of the goals attained so far, while setting ground for future work.

Being an interdisciplinary area, there are key concepts related to both software testing and evolutionary algorithms that must be introduced; these subjects are addressed in Chapter 2. In Chapter 3 related work is reviewed, with the topic of object-oriented evolutionary testing receiving special attention.

In Chapter 4, our approach to the evolutionary testing of object-oriented software is described in detail. The work performed so far allowed us to develop a prototype of a test case generation tool, which we baptized with the name *eCrash*; its framework is outlined in Section 4.2.

Experiments have been carried out and quality solutions have been found, proving the pertinence of our approach and encouraging further research; Chapter 5 describes the cases studies implemented so far, and discusses the results obtained in terms of their impact and relevance.

Still, several open problems persist in the area of search-based test case generation, and further advances must still be made in order to achieve full automation. Chapter 6 sets ground for future work and presents some topics for research. Finally, Chapter 7 resumes the key ideas and contributions of this work.

Chapter 2

Background

This Chapter provides background information on the topics addressed during the remaining of this document.

The following Section overviews software testing, by introducing different test approaches from the point of view of test case design (structural and functional), techniques employed to assemble the metrics required for test evaluation (static and dynamic analysis) and possible levels of testing (regression testing, integration testing, etc.). Next, special attention is paid to unit-testing of object-oriented programs and its terminology.

Section 2.2 presents key concepts related with evolutionary algorithms, starting by briefly exploring the genetic algorithm and genetic programming paradigms, and finally focusing on Strongly Typed Genetic Programming.

The last Section of this Chapter is devoted to explaining the concepts of Java Bytecode, presenting its main properties and demonstrating, by example, its most relevant aspects to our studies.

2.1 Software Testing

The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances [22]. Despite advances in formal methods and verification techniques, a system still needs to be tested before it is used. Testing remains the truly effective means to assure the quality of a software system of non-trivial complexity.

2.1.1 Structural and Functional Testing

Distinct test approaches – from the point of view of test case design – include Structural (or White-Box) Testing, Functional (or Black-Box) Testing, and

Grey-Box testing.

- *Functional Testing* is concerned with showing the consistency between the implementation and its functional specification.
- *Structural Testing* performs test case design with basis of the program structure.
- *Grey-box testing* is a combination of Black-Box and White-Box testing; in practice, many testing problems fall into this class. With this approach, tests are derived from a specification of the desired behaviour but with reference to the implementation details.

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. code, 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). A CFG is a representation, using graph notation, of all the paths that might be traversed through a program during its execution. Each node in the graph represents a basic block, i.e. a straight-line piece of code. Directed edges are used to represent jumps in the control flow.

2.1.2 Static Analysis and Dynamic Analysis

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 [9]; it focuses on the range of methods that are used to determine or estimate software quality without reference to actual executions. Techniques in this area include code inspection, program analysis, symbolic analysis and model checking.

In contrast, *dynamic analysis* involves executing the actual test object and monitoring its behaviour [14]; it deals with specific methods for ascertaining and/or approximating software quality through actual executions – i.e. with real data and under real (or simulated) circumstances. Techniques

in this area include synthesis of inputs, the use of structurally dictated testing procedures and the automation of testing environment generation.

Dynamic monitoring of structural entities can be achieved by instrumenting the test object, and tracing the execution of the structural entities traversed 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.

2.1.3 Levels of Testing

Although testing is involved in every stage of software life cycle, the testing done at each level of software development is different in terms of its nature and objectives [17].

- *Unit testing* tests individual application objects or methods in an isolated environment. It verifies the smallest unit of the application to ensure the correct structure and the defined operations.

- *Integration testing* is used to evaluate proper functioning of the integrated modules (objects, methods) that make up a subsystem. The focus of integration testing is on cross-functional tests rather than on unit tests within one module.

- *System testing* should be executed as soon as an integrated set of modules has been assembled to form the application; it verifies the product by testing the application in the integrated system environment.

- *Regression testing* ensures that code modification, bug correction, and any postproduction activities have not introduced any additional bugs into the previously tested code.

- *Usability testing* ensures that the presentation, data flow, and general ergonomics of the application meet the requirements of the intended users.

- *Stress testing* makes sure that the features of the software and hardware continue to function correctly under a pre-designed set and volume of test scenarios, in order to certify that the system can hold and operate efficiently under different load conditions.

- *Performance (or Timing) testing* measures the response times of the system to complete a task and the efficiency of the algorithms under varied conditions.

2.1.4 Unit Testing

The primary aim of unit testing is to uncover errors within a given unit (the *test object*) or, if no errors can be found, to gain confidence in its correctness [43]. In order to do so, the test object is executed in different scenarios using relevant and interesting *test cases*.

A *test set* is said to be adequate with respect to a given criterion if the entirety of test cases in this set satisfies this criterion; test set adequacy criteria include code or branch coverage, and are used to answer the question of what interesting test scenarios are and when the process of test case generation can be terminated.

Recent surveys [33] show that companies conduct unit testing on the basis of the programs structure (that is, white-box or grey-box testing), and that they want the test cases to be repeatable and also automated with respect to test execution and result checking.

2.1.5 Object-Oriented Unit Testing

In the case of object-oriented unit-testing, a sequence of method invocations that realizes a particular test scenario is required to cover the test goal, and the sequence search space is an explosive space.

Most work in testing has been done with "procedure-oriented" software in mind; nevertheless, traditional methods – despite their efficiency – cannot be applied without adaptation to object-oriented systems.

In an object-oriented system, the basic test unit is a class instead of a subprogram; hence, testing should focus on classes and objects. Testing a single class involves other classes, i.e. classes that appear as parameter types in the method signatures of the *class under test* (CUT); it is not possible to test the operations of a class in isolation, as they interact with each other by modifying the state of the object which invokes them. The transitive set of classes which are relevant for testing a particular class is called the *test cluster* for this class.

A unit test case for object-oriented software consists of a *method call sequence* (MCS), which represents the test scenario. During its execution, all objects participating in the test are created and put into a particular state by calling several instance methods for these objects.

Each test case focuses on the execution of one particular method, the *method under test* (MUT). Consequently, the entirety of adequate test cases for each method of the CUT satisfies the given adequacy criterion for the whole class.

In summary, the process of performing unit testing on object-oriented programs usually requires [42]:

- at least, an instance of the CUT;
- additional objects, which are required (as parameters) for the instantiation of the CUT and for the invocation of the MUT – and for the creation of these additional objects, more objects may be required;
- putting the participating objects 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.

Sometimes, software testing can benefit from object-oriented technology – for instance, by capitalizing on the fact that a superclass has already been tested, and by decreasing the effort to test derived classes, which reduces the cost of testing in comparison with a flat class structure.

However, the object-oriented paradigm can also be a hindrance to testing, due to some aspects of its very nature [4]:

- *Encapsulation* – in the presence of encapsulation, the only way to observe the state of an object is through its operations; there is therefore a fundamental problem of observability.
- *Inheritance* – inheritance opens the issue of retesting: should operations inherited from ancestor classes be retested in the context of the descendant class?
- *Polymorphism* – polymorphic names induce difficulties because they introduce undecidability in program-based testing. Moreover, erroneous casting (type conversions) are also prone to happen in polymorphic contexts and can lead non-easily detectable to errors.

2.2 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.

All of these methodologies try to solve problems for which no reasonable fast algorithms have been developed, and they are especially fit for optimization problems [8]. Independently of its class, any evolutionary program should possess the following attributes [25]:

- a *genetic representation* for potential solutions to the problem;
- a way to create an *initial population* of potential solutions;
- an *evaluation function* that plays the role of the environment, rating solutions in terms of their "fitness";
- *genetic operators* that alter the composition of children;
- *values for various parameters* that the genetic algorithm uses (population, size, probabilities of applying genetic operators, etc.).

2.2.1 Genetic Algorithms

Genetic Algorithms are the most well known form of Evolutionary Programming, having been conceived by John Holland [13] during the late sixties and early seventies. The term "Genetic Algorithm" comes from the analogy between the encoding of candidate solutions as a sequence of simple components and the genetic structure of a chromosome; continuing with this analogy, solutions are often referred to as *individuals* or *chromosomes*. The components of the solution are referred to as *genes*, with the possible values for each component being called *alleles* and their position in the sequence being the *locus*. The encoded structure of the solution for manipulation by the genetic algorithm is called the *genotype*, with the decoded structure being known as the *phenotype*.

Genetic algorithms maintain a *population* of solutions rather than just one current solution; in consequence, the search is afforded many starting points, and the chance to sample more of the search space than local searches. The population is iteratively *recombined* and *mutated* to evolve successive populations, known as *generations*. Various selection mechanisms can be used to decide which individuals should be used to create offspring for the next generation; key to this is the concept of the *fitness* of individuals.

The idea of selection is to favour the fitter individuals, in the hope of breeding better offspring; however, too strong a bias towards the best individuals will result in their dominance of future generations, thus reducing diversity and increasing the chance of premature convergence on one area of the search space. Conversely, too weak a strategy will result in too much

exploration, and not enough evolution for the search to make substantial progress.

Traditional genetic algorithm *operators* include selection, crossover, and mutation [13]:

- *Selection* (or Reproduction) is the process of copying the individuals which are going to participate in the posterior crossover phase. They are chosen according to their fitness value; selection methodologies include Fitness-Proportionate Selection, Linear Ranking or Tournament Selection;

- *Crossover* is the procedure of mating the members of the new population, in order to create a new set of individuals. As genetic material is being combined, new genotypes will be produced;

- *Mutation* modifies the values of one or several genes of an individual.

2.2.2 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. Its foremost objective is to instruct the computer on what we want it to perform [8].

In most genetic programming approaches, the programs are represented using *tree genomes* – mostly due to the influence of Koza's work [15]. The leaf nodes are called *terminals*, whereas the non-leaf nodes are called *non-terminals*. Terminals can be inputs to the program, constants or functions with no arguments; non-terminals are functions taking at least one argument. The *function set* is the set of functions from which the genetic programming system can choose when constructing trees. A set of programs is manipulated by applying reproduction, crossover and mutation until the optimum program is found or other termination criteria is met.

2.2.3 Strongly Typed Genetic Programming

The nodes of a Genetic Programming tree can be typed or non-typed: when using *non-typed* nodes, the functions of the function set are able to accept every conceivable argument; if using a *typed* mechanism 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 two individuals.

Genetic programming is a powerful method for automatically generating computer programs via the process of natural selection. However, one serious constraint on the user-defined terminals is called "closure" – i.e. all the non-terminals must accept arguments of a single data type and return values of the same data-type. This means that all non-terminals return values can be used as arguments for any other non-terminal.

In order to overcome this limitation, Montana [26] proposed the "Strongly Typed Genetic Programming" (STGP) paradigm. In STGP, variables, constants, arguments and returned values can be of any data type, with the provision that the data type for each such value be specified beforehand. This allows the initialization process and the genetic operators to only generate syntactically correct parse trees. STGP has already been extended to support type inheritance and polymorphism [12].

2.3 Evolutionary Testing

In Evolutionary Testing – also known as "Search-Based Test Case Generation" [5], a sub-area of "Search-Based Software Engineering" [10] – computational evolutionary methods are employed for test data generation.

The test objective has to be defined numerically – i.e. the test data generation process must be transformed into an optimization problem – and suitable fitness functions, that provide guidance to the search by telling how good each candidate solution is, must be defined. The fitness values are based on the monitoring results for test data.

In the particular case of object-oriented evolutionary testing, a suitable representation of object-oriented test programs must be defined; the search space of the evolutionary search is the set of all conceivable test programs for a given test object. The hidden state is, however, a serious barrier to the evolutionary approach, because of the complexity of observing the effects of method execution; this issue is usually referred to as the *State Problem*.

2.3.1 The State Problem

The State Problem [24] 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 not available to external manipulation.

In procedural software this can occur through the use of the static storage class; in object-oriented languages, through variables that are protected from external manipulation using access modifiers (most notably "getter"

and "setter" methods), and the only way to change their values is through execution of statements that perform assignments to them.

2.4 Java Bytecode

Java Bytecode is an assembly-like language that retains much of the high-level information about the original source program [39]. 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.

To understand the details of the Bytecode, a preliminary discussion on how a Java Virtual Machine (JVM) works regarding the execution of the Bytecode [18] must take place. A JVM is a stack-based machine. Each thread has a JVM stack which stores frames. A frame is created each time a method is invoked, and consists of an operand stack, an array of local variables, and a reference to the runtime constant pool of the class of the current method.

The *array of local variables* contains the parameters of the method and the values of the local variables. The size of the array of local variables is determined at compile time and is dependent on the number and size of local variables and formal method parameters. The parameters are stored first, beginning at index 0. If the frame is for a constructor or an instance method, the `this` reference is stored at location 0; location 1 contains the first formal parameter, location 2 the second, and so on. For a static method, the first formal method parameter is stored in location 0, the second in location 1, and so on.

The *operand stack* is a last-in-first-out (LIFO) stack used to push and pop values. Its size is also determined at compile time. Certain opcode instructions push values onto the operand stack; others take operands from the stack, manipulate them, and push the result. The operand stack is also used to receive return values from methods.

For example, in Figure 5.2 (Section 5.1), the `aload_1` instruction at location 0 pushes the value from the index 1 of local variable table onto the operand stack – i.e. it pushes the parameter `cfg` of the `econfigure` method onto the top of the operand stack (a reference to an object of type `Config`).

The `invokevirtual` instruction at location 1 invokes the instance method `getSignalCount` on the object `cfg` (popped from the top of the operand stack); the value returned by this method is pushed onto the top of the operand stack. The `iconst_5` instruction loads the integer value onto the top of the operand stack. At this point, the operand stack contains two values: the integer 5 on top, and the value returned by the `getPort` on the bottom. The `if_icmple` opcode loads both those values from the operand stack, and compares them: if 5 is lower than or equal to the value returned from the `getPort` method, instruction flow is transferred to instruction 18.

In fact, as the analysis of this example attests, Bytecode instructions contain enough information for coverage criteria to be applied at the Bytecode level. In addition, it can be regarded as an intermediate language, so the analysis performed at this level can be mapped back to the high-level language that generated the Bytecode.

Chapter 3

Related Work

This Chapter presents relevant work in the area of evolutionary testing, focusing on the approaches that employ structural testing techniques and address the object-oriented paradigm.

Xanthakis *et al.* [44] presented what is considered to be the first application of heuristic optimization techniques for test-data generation. Random testing was firstly employed to generate test-data, with the intention of achieving as much structural coverage as possible; then, a genetic algorithm was used to fill any gaps.

A first approach to the field of evolutionary testing of object-oriented software was presented [37]; 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 possible solutions 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 [19]. 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 branches enclosed in private/protected methods without violating encapsulation, call chain analysis on class call graphs was introduced.

In [42] the focus was put 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 does not prevent the generation of individuals which cannot be decoded into test programs without errors; the fitness function uses different penalty mechanisms in order to penalize invalid sequences and to guide the search towards regions that contain valid sequences. Due to the generation of invalid sequences, the approach lacked efficiency for more complicated cases.

In [36], a methodology for creating test software for object-oriented systems using a genetic programming approach was proposed. Experiments were carried out on five different classes. The author states that this methodology is advantageous over the more established search-based test-case generation approaches because the test software is represented and altered as a fully functional computer program. However, it is pointed out that the number of different operation types is quite limited, and that large classes which contain many methods will lead to huge hierarchical trees.

In [43] an approach in which potential solutions were encoded using a STGP methodology was presented, with MCS 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 validity; the usage of STGP preserves validity throughout the entire search process, with only compilable test cases being generated. 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 [41].

The methodology proposed by Wappler *et al.* [43, 41] yielded very encouraging results. For a custom-tailored test cluster, the set of generated test cases achieved full (100%) branch coverage: during the search, 11966

test programs were generated and evaluated, and the resulting test set contained 3 test cases; a control run, in which random test cases where produced for comparison purposes, stopped after having evaluated 43233 test programs (in accordance to the specified termination criteria), and the generated test set achieved a coverage of 66%. In a more complex scenario, four classes where tested and full coverage was achieved for all of the test objects.

Lately, Arcuri *et al.* [2, 1, 34] have developed work focused on the testing of Container Classes (e.g. Vector, Stack and Red-Black Tree). Besides analysing how to apply different search algorithms (Random Search, Hill Climbing, Simulated Annealing, Genetic Algorithms, Memetic Algorithms and Estimation of Distribution Algorithms) to the problem and exploiting the characteristics of this type of software to help the search, more general techniques that can be applied to object-oriented software were studied – such as an improved branch distance that solves an issue regarding the evaluation of conjunctions of predicates.

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, however, already studied in different scenarios. In [7] an attempt to automate the unit-testing of object-oriented programs is described. A functional approach for investigating the use of genetic algorithms for test data generation is employed, and program specifications written in JML are used for test result determination. The JML compiler was extended to make Java Bytecode produce test coverage information.

In [27] the layout of a symbolic JVM, which discovers test cases using a definable structural coverage criterion with basis on static analysis techniques, is described. The Bytecode is executed symbolically, and the decision whether to enter a branch or throw an exception is based on the earlier constraints, a constraint solver and current testing criterion. The symbolic JVM has been implemented in a test tool called GlassTT. This work, however, does not address exception-related and method interaction-related criteria, and only procedural software scenarios are described.

Interesting review articles on the topic of evolutionary testing include [23, 21, 5, 3], which overview the meta-heuristic techniques that have been used in software test data generation, such as Hill Climbing, Simulated Annealing and – most interestingly – Evolutionary Algorithms. Namely, some of the

achievements in automating test data generation in the areas of structural testing, functional testing, and grey-box testing have been summarized.

In [3], particularly, several issues in the current state-of-art of test data generation for object-oriented software are pinpointed, namely:

- little work has been done using optimisation algorithms;
- empirical tests have always been done on very small clusters of classes, which reduces the reliability of the results;
- there is no common benchmark cluster which can be used to test and compare different techniques;
- there are no comparisons between different optimisation algorithms on the testing of the same classes;
- no theoretical work on test data generation for object-oriented software exists – all articles are of empirical nature.

When comparing evolutionary-based approaches over random testing [21] several prominent advantages arise, which include: less need for human analysis, as the evolutionary algorithm pre-analyses the software in accordance to the fitness function; the ability to automatically test combinations of suspicious parameters; and the possibility of finding combinations of inputs that lead to a more severe fault behaviour. Drawbacks include the difficulty of detecting solitary errors ("needles in a haystack") with greater efficiency than random testing, and the impossibility of guaranteeing code coverage in black-box testing.

Chapter 4

Methodology and Framework

This Chapter presents our strategy for employing evolutionary algorithms for the automatic generation of unit-test cases for third-party object-oriented software.

The ideas that lead to this approach were greatly inspired by the previous works of [43, 41]. Their proposals included representing MCS as STGP trees, which are able to express the call dependences of the methods that are relevant for a given test object. This encoding is especially suited, as it 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. This means that only *compilable* test cases – i.e. test cases that do not throw exceptions during the compilation process – are generated.

Our conceptualization of the problem, however, involves performing static analysis and instrumentation of the test objects at the Java Bytecode tier; it is, in fact, possible to extract coverage metrics and derive various structural criteria without access to the source code of the program under test [40, 39]. This strategy broadens the scope of applicability of our approach, given that the test object's source code is often unavailable; it allows us, for example, to perform structural testing on third-party Java components.

The most pressing challenge to be addressed by researchers in this field is, however, the state problem; strategies for the evaluation of test cases must allow both the exploration and the exploitation of the search space if full coverage is to be attained. We propose tackling this particular hindrance by defining weighted CFG nodes, and having their weights being reevaluated each generation. This novel approach actually allows the search to consider unfeasible test cases at certain points of the search process, thus favouring diversity.

This research also hopes to contribute to increase the level of automation

of test case generation. In order to do so, the task of performing static analysis on the test object is of paramount importance, since the dependences of the test cluster and the constraints of the search space must be specified beforehand. With our approach, these are encoded in the function set; unsuitable definition of the terminal and non-terminal nodes, data-types and functions may result in the impossibility of achieving full coverage or, at least, of accomplishing it in a reasonable amount of computational time.

Our proposal includes applying existing methodologies [16, 30] for reducing the input domain of native data type values; on-going work is also focused on proposing strategies for Input Domain Reduction and Search Domain Sampling to reference data types.

Thus far, the focus of this project was put on developing the eCrash prototype tool for generating test data by employing evolutionary search techniques; the following Section starts by providing an in-depth description of the methodology, while Section 4.2 outlines the scheme of the framework. The experiments described in Chapter 5 complement this overview by illustrating the process.

4.1 Methodology

Figure 4.1 summarizes the main phases of the process, which involves the static analysis and instrumentation of the test object, the reevaluation of the CFG nodes' weights, the tasks of generating and evolving test cases, and the test case evaluation phase. The Subsections that follow describe these procedures in detail.

4.1.1 Static Analysis and Intrumentation

The first phase is that of performing static analysis on test objects's Java Bytecode; it is at this step that the test cluster, the function set and the CFG 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 transitive set of classes which are relevant for testing the MUTs are computed so as to define the test cluster. Next, the Extended Method Call Dependence Graph (EMCDG) [43], which describes the method call dependences involved in the test case construction, is build with basis on the

```
1. Static Analysis and Instrumentation Phase
 1.1. Test Object Analysis
  1.1.1. Test Cluster Definition
  1.1.2. Function Set Definition
  1.1.3. CFG Definition
  1.2.4. Parameter and Function Files Generation
 1.2. Test Object Instrumentation
2. foreach MUT
 2.1. Weight Initialization
 2.2. foreach Generation
  2.2.1. Weight Reevaluation Phase
  2.2.2. foreach Individual
   2.2.2.1. Test Case Generation Phase
    2.2.2.1.1. foreach STGP tree
     2.2.2.1.1.1. STGP tree linearization
     2.2.2.1.1.2. MCS generation
    2.2.2.1.2. Test Case Generation
    2.2.2.1.3. Test Case Compilation
   2.2.2.2. Test Case Evaluation
    2.2.2.2.1. Test Case Execution
    2.2.2.2.2. Structural Event Tracing
    2.2.2.2.3. Feasible/Unfeasible Test Case Evaluation
    2.2.2.2.4. Individual's Fitness Definition
```

Figure 4.1: Methodology overview.

test cluster. Finally, the EMCDG is evaluated in order to define the function set.

For the definition of terminal nodes representing native data types in the function set, the Ballista fault injection methodology [16] is employed. With the Ballista methodology, testing is performed by passing combinations of *acceptable*, *boundary* and *exceptional* inputs as parameters to the test object – with the intentions of *sampling* the search space for native data types and *reducing* the input domain, which has been proved to improve results in many cases [11, 30].

This strategy is emulated by identifying the definition of constants in the test object's Bytecode; these values are considered to be potential boundaries for numerical condition evaluation, and hence they – and their immediate neighbours – are included as terminal nodes in the function set. The rationale for this heuristic is the perception that this constitutes a common programming pattern; the experiment described in Section 5.1 will help illustrating this procedure.

CFGs are used as the underlying model for program representation, and

are built with basis on the information extracted from the Java Bytecode of the test object; assessing the quality of test cases involves identifying the CFG nodes traversed in the MUT. The CFG building procedure involves grouping Bytecode instructions into a smaller set of *Basic Instruction CFG nodes* and *Call CFG nodes*, with the intention of easing 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.

Finally, the test object's Bytecode is instrumented for basic block analysis and structural event dispatch, so as to enable the observation of the CFG nodes traversed during a given program execution.

4.1.2 Test Case Generation

Test cases are represented as STGP individuals; individuals are implemented as forests of STGP trees (Figure 4.2), with each individual containing a number of trees equal to the number of arguments of the MUT – i.e. each STGP tree provides an object that will be used as an argument for MUT's call.

Each tree subscribes to a function set which defines the STGP nodes legally permitted in the tree. Similarly, a STGP node is permitted to be the root if its return value type symbol matches the return value type symbol of the tree.

The first step involved in the generation of the test cases' source-code is the linearization of the trees using a depth-first transversal algorithm. The tree linearization process yields the ordered MCS. Source-code generation is performed by translating the linearized GP trees into MCS, using the method signature information encoded into the Function Files that correspond to each GP node.

Figure 4.3 depicts the MCS obtained by the translation of the STGP tree shown in Figure 4.2; Figure 4.4 presents an example of a test case generated by the eCrash tool for the case study depicted in Section 5.2.

4.1.3 Test Case Evaluation

Metaheuristic algorithms require a numerical formulation of the test goal, from which a fitness function can be derived. The purpose of the fitness function is to guide the search into promising, unevaluated areas of the search space [11].

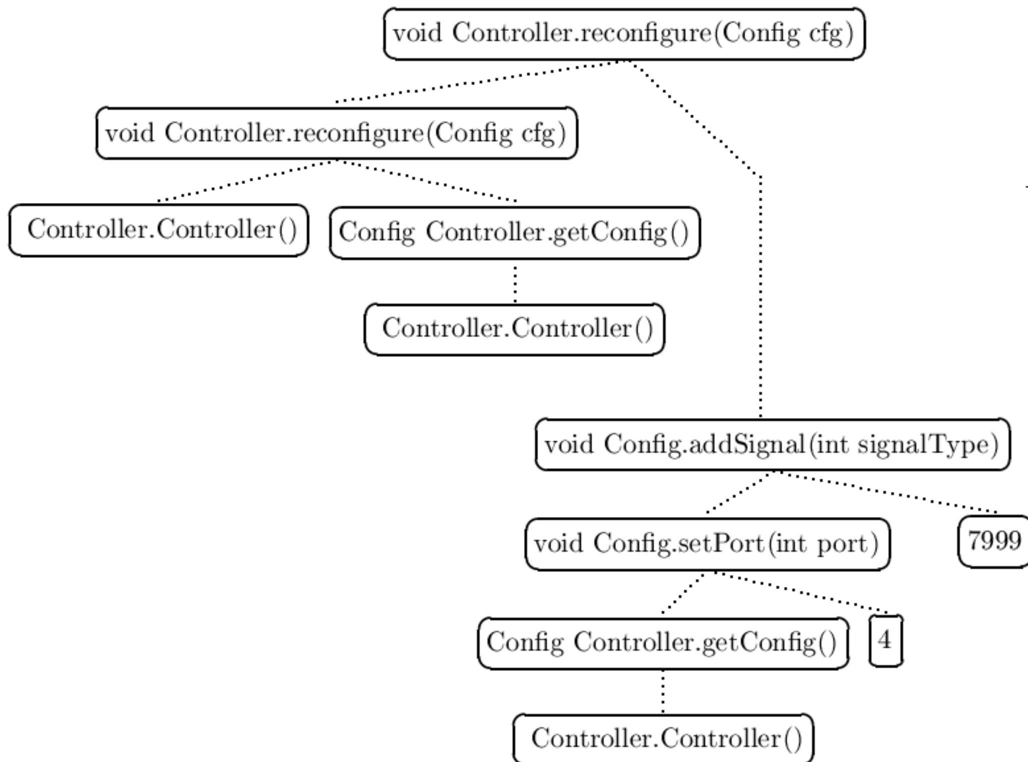


Figure 4.2: Genotype example – Strongly Typed Genetic Programming tree.

```
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 4.3: Phenotype example – Method Call Sequence.

```
public class GOI4 {
 static int seqLen = 0, exInd = 0;
 public static void main(String[] args)
   throws TCGeneException {
  Stack stack0 = paramGenerator_Stack0();
  Object object1 = paramGenerator_Object1();

  if (exInd != seqLen) { // unfeasible
   throw new TCGeneException(exInd, seqLen);
  } else { // feasible
   stack0.search(object1); // call to MUT
  }
 }

 private static Stack
   paramGenerator_Stack0() {
  Stack parameter = null;
  int c = 0;
  seqLen += 7;

  try {
   Stack stack0 = new Stack(); c++;
   String string1 = ``HelloWorld!''; c++;
   int int2 = stack0.search(string1); c++;
   Object object3 = stack0.pop(); c++;
   Object object4 = stack0.pop(); c++;
   Object object5 = stack0.peek(); c++;
   Object object6 = stack0.peek(); c++;

   parameter = stack0;
  }catch (RuntimeException e) {
   System.err.println(``paraExInd:''+c);
  } finally {
   exInd += c;
  }

  return parameter;
 }

 private static Object
    paramGenerator_Object1() {
  Object parameter = null;
  int instcount = 0;
  seqLen += 7;

  try {
   Stack stack0 = new Stack(); c++;
   String string1 = ``HelloWorld!''; c++;
   int int2 = stack0.search(string1); c++;
   boolean boolean3 = stack0.empty(); c++;
   Object object4 = stack0.peek(); c++;
   Object object5 = stack0.pop(); c++;
   Object object6 = stack0.peek(); c++;

   parameter = object6;
  }catch (RuntimeException e) {
   System.err.println(``paramExInd:''+c);
  } finally {
   exInd += c;
  }

  return parameter;
 }
}
```

Figure 4.4: Example test case.

With our approach, the quality of a given test case is related to the CFG nodes of the MUT which are the targets of the evolutionary search at the current stage of the search process. Test cases that exercise less explored (or unexplored) CFG nodes and paths must be favoured, with the objective of attaining the primary goal of the test case generation process – finding a set of test cases that achieves full structural coverage of the test object.

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 structural entities transversed in the MUT because the final instruction of the MCS is not reached. Test cases can thus be separated in two classes:

- *feasible* test cases are effectively executed, and terminate with a call to the MUT;
- *unfeasible* test cases terminate prematurely because a runtime exception is thrown by an instruction of the MCS.

As a general rule, longer and more intricate test cases are more prone to throw runtime exceptions; however, complex method call sequences are often needed for defining elaborate state scenarios and transversing certain *problem nodes*. If unfeasible test cases are blindly penalized, the definition of elaborate state scenarios will be discouraged.

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.

Additionally, the weights of CFG nodes are reevaluated at the beginning of every generation using a stigmergic process – i.e. nodes which are being recurrently traversed in previous generations and/or lead to uninteresting paths are penalised.

Weight Reevaluation

Let N be the set of Basic Block nodes of a given CFG graph; then, each CFG node $n \in N$ represents a linear sequence of computations (i.e. Bytecode instructions) of the MUT, and each CFG edge e_{ij} represents the transfer of the execution control of the program from node n_i to the node n_j.

Conversely, n_j is a successor node of n_i if an edge e_{ij} between the nodes n_i and n_j exists. The set of successor nodes of n_i is defined as $N_s^{ni}, N_s^{ni} \subset N$.

The weight of transversing node n_i is identified as W_{ni}. At the beginning of the evolutionary search the weights of nodes are initialized with a predefined value W_{init}. W_{max} corresponds to the maximum value for the weight existing in N.

The CFG nodes' weights are reevaluated at the beginning of every generation according to Equation 4.1.

$$W_{ni} = (\alpha W_{ni}) \left(\frac{hitC_{ni}}{|T|} + 1 \right) \left(\frac{\sum_{x \in N_s^{ni}} W_x}{|N_s^{ni}| \times \frac{W_{init}}{2}} \right) \tag{4.1}$$

The $hitC_{ni}$ parameter is the "Hit Count", and contains the number of times a particular CFG node was exercised by the test cases of the previous generation. T represents the set of test cases produced in the previous generation.

The constant value $\alpha, \alpha \in]0, 1]$ is the "weight decrease constant".

After being reevaluated, weights of all the nodes are normalized in accordance to Equation 4.2:

$$W_{ni} = \frac{W_{ni} \times W_{init}}{W_{max}} \tag{4.2}$$

Evaluation of Feasible Test Cases

Let t be the test case being currently evaluated. For feasible test cases, the fitness is computed by with basis on their trace information; relevant trace information includes the the "Hit List" – i.e. the set $H_t, H_t \subseteq N$ of transversed CFG nodes.

The fitness of feasible test cases is evaluated in accordance to Equation 4.3:

$$Fitness_{feasible}(t) = \frac{\sum_{h \in H_t} W_h}{|H_t|} \tag{4.3}$$

Evaluation of Unfeasible Test Cases

For unfeasible test cases, the fitness of the individual is calculated in terms of the distance between the "runtime exception index" $exInd_t$ (i.e. the position of the method call that threw the exception) and the "method call sequence length" $seqLen_t$. Also, an "unfeasible penalty constant" value β is added to the final fitness value, so as to penalise unfeasibility.

Equation 4.4 depicts the formula for evaluating unfeasible test cases.

$$Fitness_{unfeasible}(t) = \beta + \frac{(seqLen_t - exInd_t) \times 100}{seqLen_t} \tag{4.4}$$

The $seqLen_t$ is the number of instructions that make the MCS of test case t; the exception index $exInd_t$ parameter refers to the sum of instructions actually executed in t before a runtime exception is thrown.

With this methodology, and depending on the value of β and on the fitness of feasible test cases, unfeasible test cases may be selected for breeding at certain points of the evolutionary search, thus favouring the diversity and complexity of method call sequences. This will happen if feasible test cases always transverse recurrently hit nodes, thus increasing their weight and worsening the fitness of the corresponding test cases.

4.2 Framework Overview

The focus of this project was put on developing the eCrash prototype tool for generating test data by employing evolutionary search techniques; this tool is composed by the following main modules:

- *Automatic Test Object Analyzer (ATOA)* – the test object analysis is performed by this module; it's main tasks are those of defining the test

cluster, and automatically generating Parameter and Function Files which define the Function Set and contain the constraints of the STGP algorithm.

- *Static Analysis and Instrumentation Module (SAIM)* – executes the task of building the CFG and instrumenting the Bytecode of the test object.
- *Test Case Generator Module (TCGM)* – evolves and linearizes STGP trees, and constructs test cases.
- *Test Case Evaluator Module (TCEM)* – evaluates test cases and provides the TCGM with feedback on their quality.

A graphical outline of the eCrash tool's framework is depicted in Figure 4.5.

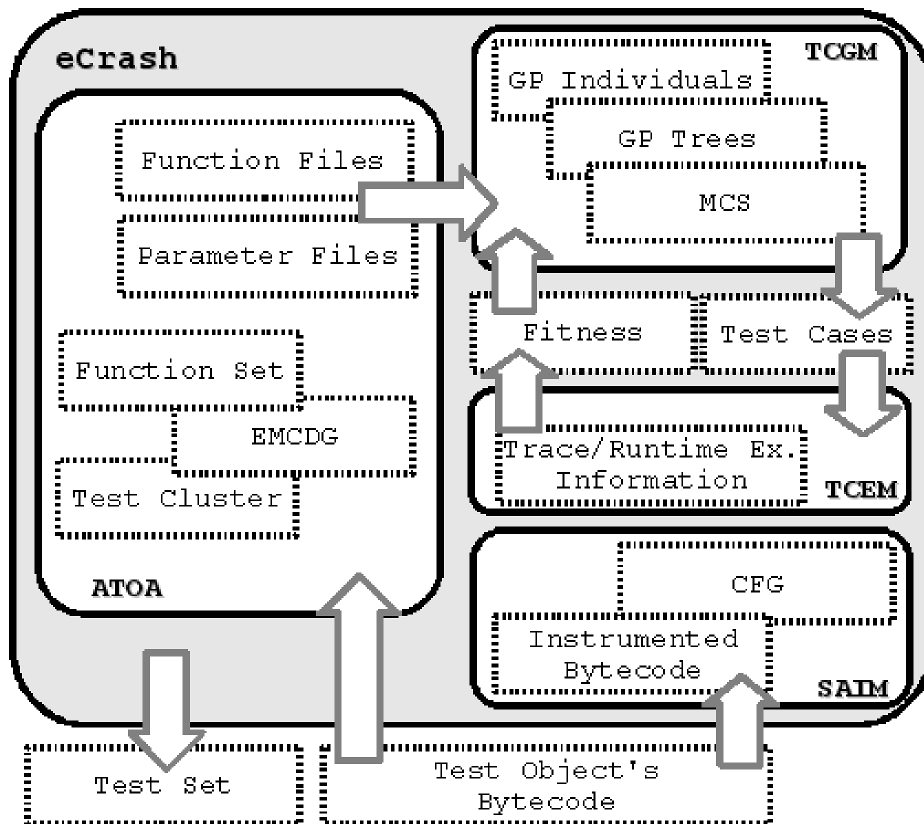


Figure 4.5: Framework overview.

The tasks carried out by ATOA and TCGM are performed off-line; they both receive the test object's Bytecode as an input, and yield the files and

structures required for the test cases to be evolved – namely the Parameter and Function files, the CFG of the test object, and the instrumented test object.

The TCGM and TCEM employ the information received by the above-mentioned modules to perform the search-based generation of test cases. Finally, when a predefined termination criteria is met (i.e. full structural coverage is achieved or a maximum number of generations is reached), the eCrash tool returns the test set generated for the test object.

The process of CFG building, bytecode instrumentation and event tracing is achieved with the aid of Sofya [14], 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 analises using the outputs generated by its components (statistics, coverage reports, ...) and to visualize the trace information produced by the executions of instrumented programs.

For evolving the set of test cases, the Evolutionary Computation in Java (ECJ) package [20] 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 files.

Chapter 5

Experimental Studies

In order to validate and clarify our approach, experiments were performed on two distinct test objects:

- The custom-made "Controller & Config" test cluster proposed in [41]. This test cluster encompasses 2 classes and 7 public methods, but only the `Controller.reconfigure(Config)` method was used as a MUT.
- The `Stack` class of the `java.util` package of JDK 1.4.2. The public API of this class is composed by 5 public methods, and all of them were subjected to the test case generation process.

The "Controller & Config" test object was used throughout the development process of the eCrash tool and, in addition to providing interesting data for research, it constituted a precious aid during the process of debugging. The main goals of the experiment were those of demonstrating and proving the feasibility of the approach, while gathering preliminary results with which to fine-tune our methodology.

The `Stack` case study was developed with the following intentions: experimenting with different configurations for the probabilities of the evolutionary operators mutation, reproduction and crossover; investigating the impact of using distinct values for the "weight decrease constant" α (Equation 4.1) and the "unfeasible penalty constant" β (Equation 4.4) – which will henceforth be referred to as *test case evaluation parameters*. Additionally, it allowed demonstrating the applicability of the approach to a "real world" problem.

The Sections that follow describe both these experiments in detail: firstly, the configurations and parameterizations defined for each case study are described; secondly, the results obtained are presented. This Chapter ends with a discussion on the main achievements attained, while pointing out occasional flaws in our methodology and/or framework.

5.1 Case Study: Controller & Config

In this case study, the simple test cluster proposed in [41] is employed; its source code is reproduced in Figure 5.1.

The `Controller.reconfigure(Config)` public method was used as the MUT; the Bytecode instructions for this method are depicted in Figure 5.2.

```
public class Controller {
 protected final static int MAX_SIGNALS = 5;
 protected final static int MIN_PORT = 8000;
 protected final static int MAX_PORT = 8005;

 private Config cfg = null;
 private int[] signals = null;

 public Controller() {
  cfg = new Config(-1);
  signals = new int[cfg.getSignalCount()];
 }

 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()];
 }

 public int retrieve(int signal) {
  if( signal<0 || signal>signals.length-1 )
   throw new IllegalArgumentException
            (``Invalid signal.'');
  return signals[signal];
 }
 public Config getConfig() {
  return cfg;
 }
}

public class Config {
 private Vector signals;
 private int port;

 public Config(int port) {
  this.port = port;
  signals = new Vector();
  addSignal(0);
 }

 public void addSignal(int signalType) {
  signals.add(new Integer(signalType));
 }

 public int getSignalCount() {
  return signals.size();
 }

 public int getPort() {
  return port;
 }

 public void setPort(int port) {
  this.port = port;
 }
}
```

Figure 5.1: Source code of the “Controller & Config” test object.

The static analysis phase yielded the function set depicted in Table 5.1. It should be noted that the integer value terminal set Tn = {`Integer.MAXVALUE`, `Integer.MINVALUE`, 0, 4, 5, 6, 7999, 8000, 8001, 8004, 8005, 8006} was defined as a result of the test object's Bytecode analysis, and in accordance to the Ballista methodology [16] described in Section 4.1.1.

The Ballista methodology was emulated by identifying the definition of constants in the MUT's Bytecode (Figure 5.2). 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

Function Name	**Return Type**	**Child Types**
	Controller class	
`Controller()`	Controller	-
`void reconfigure(Config cfg)`	Controller	Controller, Config
`void reconfigure(Config cfg)`	Config	Controller, Config
`void printSignals()`	Controller	Controller
	Config class	
`public Config()`	Config	-
`void addSignal(int signalType)`	Config	Config, int
`int retrieveSignal(int signalIndex)`	Config	Config, int
`int retrieveSignal(int signalIndex)`	int	Config, int
`int getSignalCount()`	Config	Config
`int getSignalCount()`	int	Config
	Integer Value Terminal Set (Tn)	
`Integer.MAXVALUE`	int	-
`Integer.MINVALUE`	int	-
`0`	int	-
`4`	int	-
`5`	int	-
`6`	int	-
`7999`	int	-
`8000`	int	-
`8001`	int	-
`8004`	int	-
`8005`	int	-
`8006`	int	-

Table 5.1: Function Set for the “Controller & Config” case study.

```
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
```

Figure 5.2: Bytecode of the `reconfigure` method.

values were considered to be potential boundaries for numerical condition evaluation, and hence their inclusion and that of their immediate neighbours (4, 6; 7999, 8001; 8004, 8006) into the Tn set. 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 5.3. The bytecode instructions of the MUT (Figure 5.2) were mapped to the CFG nodes in accordance to Table 5.2; relevant information includes the mapping between the MUT's Bytecode instructions, and Basic Instruction and Call CFG node types and sub-types, and CFG node numbers.

Attaining full structural coverage of the MUT requires the transversal of all the Basic Instruction (4, 5, 8, 11, 12, 15) and Call (2, 6, 9, 13) CFG nodes.

5.1.1 Setup

The test case evaluation parameters were defined as follows: the MUT's CFG nodes were initialized with $W_{init} = 200$; the "weight decrease constant" α was set to 0.9, and the "unfeasible penalty constant" β was defined as 150.

ECJ was configured using a single population of 5 individuals. Each individual was composed by 2 STGP trees – with the first returning an instance of `Controller` (required to call the `reconfigure` instance method), and the second returning an object of type `Config` (required as an argument for the `reconfigure` method).

For the generation of individuals a multi-breeding pipeline was used,

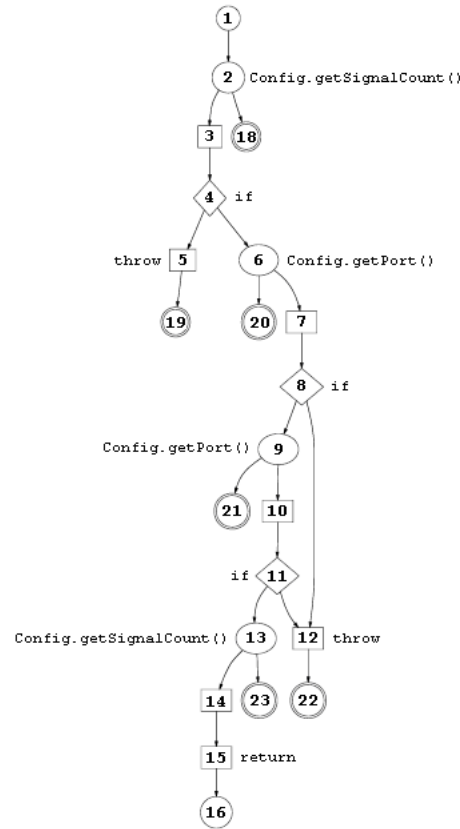


Figure 5.3: Control-Flow Graph of the `reconfigure` method.

Initial Bytecode	**Final Bytecode**	**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	11
48	55	Call		13
58	63	Basic	Return	15

Table 5.2: Mapping table between the Bytecode instructions and the Control-Flow Graph nodes for the `reconfigure` method.

which stored 3 child sources; each time an individual had to be produced, one of those sources was selected with a predefined probability. The available breeding pipelines were the following:

- *Reproduction pipeline* – simply makes a copy of the individuals it receives from its source.
- *Crossover pipeline* – performs a strongly-typed version of "Subtree Crossover" [15]. Two individuals are selected, and a single tree is chosen in each such that the two trees have the same constraints; then a random node is chosen in each tree such that the two nodes have the same return type, and the swap is performed.
- *Mutation pipeline* – implements a strongly-typed version of the "Point Mutation" [15]. An individual is selected, a random node is selected, and the subtree rooted at that node is replaced by a new valid tree.

The probability of choosing the crossover, mutation and reproduction were given the values of 0.6, 0.2 and 0.2, respectively.

The selection method employed was *Tournament Selection* with a size of 2.0, which means that first 2 individuals (out of 5) are chosen at random from the population, and then the one with the best fitness is selected. STGP trees were grown (for the purposes of initialization and mutation) using the "Ramped Half-And-Half" tree building method described in [15].

The search stopped if an ideal individual was found or after 200 generations. The remaining configurations used were the Koza-style parameters defined in ECJ by default.

It should be noted that the probabilities for the evolutionary operators and the configurations of the test case evaluation parameters were defined empirically, as no experiments had yet been implemented that could provide guidance for their selection; the case study described in Section 5.2 will, however, provide guidelines for their choice.

5.1.2 Results

Full structural coverage was achieved in all of the runs in an average of 27.6 generations (Table 5.3). 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).

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 traverse problem CFG node 5. In fact, the CFG node 5 is paradigmatic

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

Table 5.3: Number of generations required to attain full coverage.

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 signal)` method of the `Config` object that will be used as an argument in the MUT call 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 5.4). 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 β, thus allowing for unfeasible test cases to be selected for breeding.

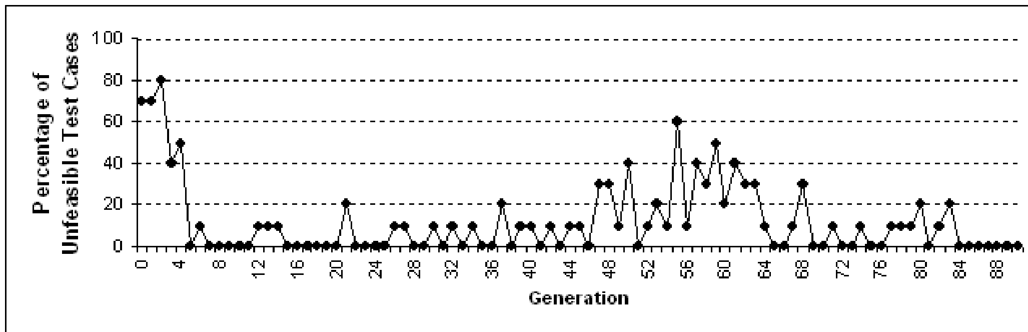


Figure 5.4: Percentage of unfeasible test cases per generation.

The usefulness of the our methodology is particularly visible if the results are compared to those obtained using random search (Table 5.3). In order to 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 was not 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 traversed; in the remaining 4 runs, the results yielded 70% code coverage – CFG nodes 5, 13 and 15 were not exercised.

5.2 Case Study: Stack

This case study was developed with the intentions of demonstrating the applicability of the approach to a "real world" problem, while experimenting with different parameters and configurations; another interesting property of the `Stack` class is that, being a container class, it contains explicit state, which is only controlled through a series of method calls.

All of the public methods of this class were subjected to the automated test case generation process, namely:

- `boolean empty()`
- `Object peek()`
- `Object pop()`
- `Object push(Object item)`
- `int search(Object o)`

Two distinct studies were made:

- the *Probabilities of Operators* case study was performed with the intention of assessing the implications of evolutionary operators' probabilities on the test case generation process;
- the *Evaluation Parameters* case study took place with the objective of analyzing the impact of the test case evaluation parameters α and β on the evolutionary search.

The static analysis process yielded the function set depicted in Table 5.4.

Function Name	Return Type	Child Types
`boolean empty()`	boolean	Stack
`boolean empty()`	Stack	Stack
`Object peek()`	Object	Stack
`Object peek()`	Stack	Stack
`Object pop()`	Object	Stack
`Object pop()`	Stack	Stack
`Object push(Object item)`	Object	Stack, Object
`Object push(Object item)`	Stack	Stack, Object
`int search(Object o)`	int	Stack, Object
`int search(Object o)`	Stack	Stack, Object
`int search(Object o)`	Object	Stack, Object
`Stack()`	Stack	-
`Object()`	Object	-
`''HelloWorld!''`	Object	-

Table 5.4: Function Set for the `Stack` case study.

5.2.1 Setup

ECJ was configured using a single population of 5 GP individuals; the number of GP trees per individual was equal to the number of child types of the MUT (Table 5.4), as each tree provided the method with an argument. The MUTs' CFG nodes were initialized with a weight W_{init} of 200. The search stopped if an ideal individual was found or after 200 generations.

For the generation of individuals, a multi-breeding pipeline similar to the one described in Section 5.1 was used, which included strongly-typed versions of "Subtree Crossover" and "Point Mutation", and a simple reproduction operator. Tournament selection, with a size of 2.0, was employed as the selection method, and trees were grown using the "Ramped Half-And-Half" tree building method.

5.2.2 Results

Probabilities of Operators

This particular experiment was performed with the intention of assessing the implications of evolutionary operators' probabilities on the test case generation process.

In order to do so, 4 distinct parameterizations of the multi-breeding

pipeline were defined, having:

1. a high probability of selecting the mutation breeding pipeline;
2. a high probability of selecting the crossover breeding pipeline;
3. a high probability of selecting the reproduction breeding pipeline;
4. equal probabilities of selecting either of the above breeding pipelines.

The "weight decrease constant" α was set to 0.9, and the "unfeasible penalty constant" β was defined as 150.

For each of the above multi-breeding pipeline parameterizations, 20 runs were executed for each of the 5 MUTs. Table 5.5 summarizes the results obtained. Relevant data includes the probabilities of choosing reproduction (r), crossover (c) and mutation (m) pipelines and, for each configuration, percentage of runs in which full coverage was achieved (*%full*) and the number of generations required attain full coverage (*#gens*).

	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

Table 5.5: Statistics for the "Probabilites of Operators" experiment.

The results depicted clearly show that the strategy of assigning balanced probabilities to the all of the breeding pipelines yields better results: this configuration was the only one in which full coverage was achieved in all of the runs (in, at most, 200 generations), and in beat all the other configurations in terms of the average number of generations required to attain it.

The worst results were obtained for the parameterization in which the reproduction breeding pipeline was given a high probability of selection. For the `Object push(Object item)` and `int search(Object o)` MUTs, which pose the most challenging state problems, 43% of the runs failed to attain full coverage within 200 generations.

Evaluation Parameters

In this experiment, different combinations of values for the α and β parameters were tried out, with the intention of analyzing the impact of the test case evaluation parameters on the evolutionary search particular.

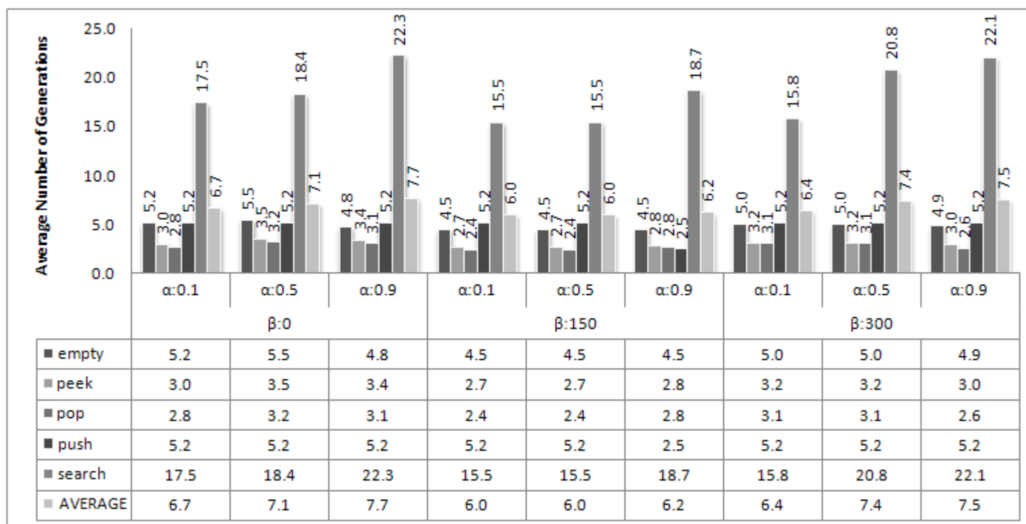
Namely, the following values were used:

- α – 0.1, 0.5, and 0.9;
- β – 0, 150, and 300.

The probabilities of choosing the 3 breeding pipelines were chosen in accordance to the results yielded by the **Probabilities of Operators** case study – i.e. the probabilities for reproduction, crossover an mutation parameters were set to 0.33, 0.33 and 0.34 respectively.

All the 9 combinations of the α and β values were employed, and 20 runs were executed for each (in a total of 180 runs); full coverage was achieved in all of the runs.

The results obtained are summarized in Figure 5.5, which includes the average number of generations required to attain full coverage for each of the 5 MUTs using each combination.



	β:0			β:150			β:300		
	α:0.1	α:0.5	α:0.9	α:0.1	α:0.5	α:0.9	α:0.1	α:0.5	α: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	2.6
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

Figure 5.5: Results for the "Evaluation Parameters" experiment.

These results clearly 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 150 to β.

5.3 Discussion

Automatic test case generation using search-based techniques is a difficult subject, especially if the aim is to implement an "universal" solution that is adaptable to a wide range of test objects. Key to the definition of a good strategy is the configuration of parameters so as to find a good balance between the intensification and the diversification of the search.

With our approach, test case evaluation parameters α and β, and the evolutionary operators' selection probabilities, play a central role in the test case generation process.

The main task of the mutation and crossover operators is that of *diversifying* the search, allowing it to browse through a wider area of the search landscape and to escape local maximums; the task of *intensifying* the search and guiding it towards the transversal of unexercised CFG nodes is performed as a result of the strategy of assigning weights to CFG nodes.

Nevertheless, to strong a bias towards the breeding of feasible test cases will hinder the generation of more complex test cases, which are sometimes needed to exercise problem structures in the test object; on the other hand, if feasible test cases are not clearly encouraged, the search process will wander.

This issue was addressed by allowing the fitness of feasible test cases to fluctuate throughout the search process as a result of the impact of the α and β parameters, in order to allow unfeasible test cases to be selected at certain points of the evolutionary search.

The results described for the `Stack` case study (Section 5.2), in particular, allow us to draw some preliminary conclusions. Firstly, the assumption we made on previous studies, in which we considered $\alpha = 0.9$ as being an adequate value, was wrong. Using lower values for this evaluation parameter yields better results. Secondly, it was also possible to conclude that the strategy of assigning balanced probabilities to the selection of the mutation, crossover and breeding pipelines is better suited that that of clearly favouring the choice of one of the evolutionary operators.

On the other hand, it is possible to affirm that the strategy of assigning the value of 150 to the "unfeasible penalty constant" β shows good results. An explanation for this behaviour follows.

The worst value a CFG node can have is 200 – since the weights of CFG nodes are normalized at the beginning of each generation. If all the nodes exercised by a feasible test case have the worst possible value – because they are being recurrently exercised by test cases, i.e. because the search is stuck in a local maximum – the fitness of the corresponding test case will also be 200.

However, for a given unfeasible test case t, if $exInd_t \leq \frac{seqLen_t}{2}$ and $\beta =$

150, then $Fitness_{unfeasible}(t) \in [150, 200]$ – i.e. if the exception index of a given unfeasible test case is lower or equal to half of its MCS length, and if the value 150 is used for β, then the fitness of that test case will belong to the interval 150 to 200.

This means that, with $\beta = 150$, some *good* unfeasible test cases may be selected for breeding; conversely, if $\beta = 0$, *all* unfeasible test cases will be evaluated with relatively good fitness values, and if $\beta = 300$, *none* of the unfesible test cases will be evaluated as being interesting. The concept of good unfeasible test cases, in this context, can thus be verbalized as being a test case in which at least half of the MCS is executed without an exception being thrown.

Assigning the value $\beta = W_{init} - 50$ is, thus, a good compromise between the need to penalize unfeasible test cases and the need to consider them at some points of the evolutionary search.

It is also possible to affirm that the Ballista methodology for sampling the integer value search space effectively reduced the input domain for this particular data type; by comparing the results attained using this strategy to those yielded by the usage of a random integer value generator, we can conclude that this strategy allowed achieving full coverage within an acceptable number of generations.

Chapter 6

Future Work

There are still many enhancements to be made in order to achieve full automation, and several open problems persist in the area of search-based test case generation.

An underlying principle of our investigation involves the continuous research on the definition and parameterization of fitness functions that can effectively and efficiently overcome the state problem of object-oriented programs.

In the near future, it is also of paramount importance to perform further experimental studies on different test objects and using various evolutionary parameters, as well as comparing the results obtained with those of existing automation methodologies.

In order to do so, further developments to the ATOA module of the eCrash framework are in order. Although this module is already partly automated, some bugs persist that hinder the generation of fully functional Function Files and Parameter Files. Some minor corrections and additions to these files must still be made manually, which make the process of generating the data needed to experiment on new test objects awkward and time-costly.

The implementation of a Graphical User Interface (GUI), that enables the eCrash tool to be utilized by users unfamiliar with its intricacies, is also in the horizon. So far, only the ATOA module incorporates a visual interface.

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

- *Method Call Sequence Separation* – deals with generating independent method call sequences for each argument of the MUT.
- *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 make enable the coverage of the entire search space.

The pertinence of investigating these particular topics is addressed in the following subsections.

6.1 Method Call Sequence Separation

In preliminary approaches to problem at hand, test cases were composed by a single MCS, in a similar fashion the the methodology proposed by [43, 41].

However, alongside our work, we realized that this constituted a hindrance to the evaluation of unfeasible test cases. We thus proceeded to employing a different approach, which included using a number of STGP trees – and, consequently, generating a number of MCS – equal to the number of arguments in the MUT. This way, it is possible to evaluate the MCS that generate each argument independently, thus allowing for a more realistic definition of the $exInd_t$ (Equation 4.4).

The example described below, based on the sample test case depicted on Figure 4.4, helps exemplifying the rationale of performing MCS separation.

With our current approach, a method call sequence is generated for each argument (of types `Stack` and `Object`) by the parameter generator methods `paramGenerator_Stack0` and `paramGenerator_Object1`. However, both these method throw runtime exceptions (`EmptyStackExceptions`) at instructions 4 and 5, respectively. The exception index is thus $exInd_t = 7$ – i.e. it is equal to the number of instruction actually executed – with $seqLen_t$ being 14.

However, if a single tree was used, it would be linearized into a single MCS, and the exception index would thus be 3 and not 7, as the instructions related with setting the state of the second argument would never be called.

Our current approach obviously yields more realistic and reliable results than that of using a single MCS; however, further data must be collected and analyzed in order to support this claim.

6.2 Input Domain Reduction

For search-based test data generation, the search space is the input domain of the test object under consideration, which include the formal parameters to the function containing the structure of interest. As such, it is possible

that not every input variable will be responsible for determining whether each structure will be covered or not.

The Ballista approach technique [16] for domain reduction has been proven to be effective for native data types [30, 31], and techniques for extending this procedure for reference types are yet to be investigated; nevertheless, recent research on procedural software provides evidence to support the claim that domain reduction has significant implications for practical search-based test data generation [11].

Future work thus includes the definition of strategies for reducing the input domain of reference argument types, in a similar fashion to that described in Section 5.1 for native data types. This can be achieved by removing irrelevant elements from the function set of a given problem.

Let us consider the `Stack` case study (Section 5.2). By studying the documentation of the `Stack` class, the following observations can be made:

- the `empty` method simply tests if the stack is empty without changing the state of the stack;
- the `peek` method looks at the object at the top of the stack without removing it, and thus it does not alter the stack;
- the `push` method pushes an item onto the top of the stack, and it does not change the state of the pushed item;
- the `search` method returns the position where an object is on the stack – if it is on the stack – and it does not change the state of either the stack or the object being searched.

Therefore, the search domain for the Stack case study could effectively be reduced by employing the function set shown in Table 6.1. Five entries were removed from the original function set (Table 5.4).

However, these observations were made for demonstration purposes only. Our approach to software testing is a structural one, and it does not include considering any kind of specification, neither formal nor informal; we believe that doing so would narrow the scope of applicability and the level of automation of our testing tool.

There are, however, systematic approaches to execute this type of verification – ex. by performing “Purity Analysis” [35, 45] on the methods that compose the function set. Basically, a pure method has no externally visible side effects; purity analysis checks, for each method, what fields they potentially modify. Future research includes embedding this particular technique into our framework.

Function Name	Return Type	Child Types
`boolean empty()`	boolean	Stack
`Object peek()`	Object	Stack
`Object pop()`	Object	Stack
`Object pop()`	Stack	Stack
`Object push(Object item)`	Stack	Stack, Object
`int search(Object o)`	int	Stack, Object
`Stack()`	Stack	-
`Object()`	Object	-
`‘‘HelloWorld!’’`	Object	-

Table 6.1: Improved Function Set for the `Stack` case study.

6.3 Search Domain Sampling

The failure to acknowledge the importance of the polymorphic relationship between objects can significantly hinder the evolutionary search, and may result in the impossibility of attaining full coverage.

In order to demonstrate the reason for this claim, one more experiment was developed using the `Stack` test object (Section 5.2). The objective was that of demonstrating the importance of the polymorphic relationships of the object types when defining the function set: the “HelloWorld!” function, depicted in the last row of Table 5.4, was removed from the function set, and a battery of 20 runs was performed.

Reproduction, crossover an mutation probabilities were set to 0.34, 0.33 and 0.33 respectively, and the test case evaluation parameter were configures with $\alpha = 0.1$ and $\beta = 150$.

For the first four MUTs, the results attained were similar to those depicted in Figure 5.5. However, for the `search` method, full coverage was not achieved in any of the 20 runs.

In fact, including the default constructor `Object()` as the sole provider of `Object` data types does not suffice to achieve full coverage. The reason for this is that the `equals` method is used internally to compare argument `o` to the items in the stack; however, the `equals` method of class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-`null` reference values `x` and `y`, this method returns true if and only if `x` and `y` refer to the same object.

With our approach instances are not reused, and hence the search for argument `o` always fails.

The `String` class, however, overrides the `equals` method so that the result is true if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as the `this` object.

Therefore, the inclusion of the `Object`'s subclass `String` as an `Object` type provider enables the `equals` method to evaluate favourably.

However, the test cluster cannot include all the subclasses of the transitive set of classes which are relevant for the test object. The `Object` type is paradigmatic: all classes inherit from the `Object` class, and hence *all* classes would have to be included, which is – obviously – impossible.

Future research thus includes studying the polymorphic relationships of function set types in order to define an adequate strategy for sampling the search domain. Tuning our test case generation process for reusing instances is also in order, since some scenarios may require this particular functionality.

Chapter 7

Conclusions

This thesis presents an evolutionary approach for the structural unit-testing of third-party object-oriented software. Relevant contributions include (but are not limited to) the introduction of novel methodologies for automation and search guidance, and the presentation of the eCrash prototype test case generation tool.

With our approach, CFG nodes are weighted; additionally, their weight is dynamically reevaluated each generation, in order to cause the fitness of feasible test cases to fluctuate throughout the search process. This strategy allows unfeasible test cases to be considered at certain points of the evolutionary search – once the feasible test cases that are 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.

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 methodology for evaluating the test set includes instrumenting the Bytecode for basic block analysis and structural event dispatch, and executing the instrumented test object using the generated test cases as inputs, with the intention of collecting trace files with which to derive coverage metrics. Static analysis, instrumentation and execution tracing is performed solely with basis on the high-level information extracted from the Java Bytecode of the test object.

Throughout our work we have come across – and tackled – complex challenges. Alas, because many of them were technical problems that gravitate around the central “core” of this research, their impact on the development process may not have been efficiently mirrored in this report. Special refer-

ences should be made to the integration of the third-party components (ECJ and Sofya) into our framework, the automatic generation of syntactically correct Function and Parameter files, the "on-the-fly" compilation and execution of test programs, the translation of the information encoded in STGP individuals into fully functional test cases – and, especially, to the process of "gluing together" the whole process.

We have managed, all the same, to achieve a very high level of automation, which allowed us to experiment our prototype tool and develop pertinent case studies, attaining very encouraging results which pointed us towards future endeavours.

There is, in fact, much uncharted territory in the area of evolutionary testing, and we hope to keep contributing to the advance the state-of-the-art in years to come. The "holy grail" is, of course, the implementation of a fully functional tool that can be integrated in the software development process, providing a precious aid to the often tiresome – although incommensurably important – process of software testing.

Bibliography

[1] Andrea Arcuri and Xin Yao. A memetic algorithm for test data generation of object-oriented software. In Dipti Srinivasan and Lipo Wang, editors, *2007 IEEE Congress on Evolutionary Computation*, pages –, Singapore, 25-28 September 2007. IEEE Computational Intelligence Society, IEEE Press.

[2] Andrea Arcuri and Xin Yao. Search based testing of containers for object-oriented software. Technical Report CSR-07-3, University of Birmingham, School of Computer Science, April 2007.

[3] Andrea Arcuri and Xin Yao. On test data generation of object-oriented software. In *Testing: Academic and Industrial Conference, Practice and Research Techniques (TAIC PART)*, 2007 (to appear).

[4] Stéphane Barbey and Alfred Strohmeier. The problematics of testing object-oriented software. In M. Ross, C. A. Brebbia, G. Staples, and J. Stapleton, editors, *SQM'94 Second Conference on Software Quality Management, Edinburgh, Scotland, UK, July 26-28 1994*, volume 2, pages 411–426, 1994.

[5] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[6] Yoonsik Cheon and Myoung Kim. A specification-based fitness function for evolutionary testing of object-oriented programs. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1953–1954, New York, NY, USA, 2006. ACM Press.

[7] Yoonsik Cheon, Myoung Kim, and Ashaveena Perumandla. A complete automation of unit testing for java programs. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 290–295. CSREA Press, 2005.

[8] Francisco Fernandez de Vega. *Distributed Genetic Programming Models with Application to Logic Synthesis on FPGAs.* PhD thesis, University of Extremadura, 2001.

[9] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.

[10] Mark Harman. The current state and future of search based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.

[11] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. 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*, pages 155–164, New York, NY, USA, 2007. ACM Press.

[12] Thomas D. Haynes, Dale A. Schoenefeld, and Roger L. Wainwright. Type inheritance in strongly typed genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 359–376. MIT Press, Cambridge, MA, USA, 1996.

[13] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* The MIT Press, April 1992.

[14] A. Kinneer, M. Dwyer, and G. Rothermel. 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.

[15] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* The MIT Press, December 1992.

[16] Nathan P. Kropp, Philip J. Koopman Jr., and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Symposium on Fault-Tolerant Computing*, pages 230–239, 1998.

[17] Kanglin Li and Mengqi Wu. *Effective Software Test Automation: Developing an Automated Software Testing Tool.* SYBEX Inc., Alameda, CA, USA, 2004.

[18] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[19] Xiyang Liu, Bin Wang, and Hehui Liu. Evolutionary search in the context of object-oriented programs. In *MIC'05: Proceedings of the Sixth Metaheuristics International Conference*, 2005.

[20] Sean Luke. ECJ 16: A Java evolutionary computation library. http://cs.gmu.edu/∼eclab/projects/ecj/, 2007.

[21] Timo Mantere and Jarmo T. Alander. Evolutionary software engineering, a review. *Appl. Soft Comput.*, 5(3):315–331, 2005.

[22] John J. Marciniak, editor. *Encyclopedia of software engineering*. Wiley-Interscience, New York, NY, USA, 1994.

[23] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[24] P. McMinn and M. Holcombe. The state problem for evolutionary testing, 2003.

[25] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (2nd, extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1994.

[26] David J. Montana. Strongly typed genetic programming. Technical Report #7866, 10 Moulton Street, Cambridge, MA 02138, USA, 7 1993.

[27] Roger A. Müller, Christoph Lembeck, and Herbert Kuchen. A symbolic java virtual machine for test case generation. In M. H. Hamza, editor, *IASTED Conf. on Software Engineering*, pages 365–371. IASTED/ACTA Press, 2004.

[28] Estelle M. Phillips and Derek S. Pugh. *How to get a PhD: a handbook for students and their supervisors*. Open University Press, Milton Keynes, 1987.

[29] José Carlos Bregieiro Ribeiro, Francisco Fernandez de Vega, and Mário Zenha 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 of the 25th Brazilian Symposium on Computer Networks and Distributed Systems*, pages 143–156. Brazilian Computer Society (SBC), 2007.

[30] José Carlos Bregieiro Ribeiro, Bruno Miguel Luís, and Mário Zenha-Rela. Error propagation monitoring on windows mobile-based devices. In *LADC 2007: Third Latin-American Symposium on Dependable Computing*, volume 4746/2007 of *Lecture Notes in Computer Science*, pages 111–122. Springer Berlin / Heidelberg, 2007.

[31] José Carlos Bregieiro Ribeiro, Mário Zenha-Rela, and Francisco Fernandez de Vega. ecrash: a framework for performing evolutionary testing on third-party java components. In *JAEM CEDI 2007: Proceedings of the 1st Jornadas sobre Algoritmos Evolutivos y Metaheuristicas of the 2nd Congreso Español de Informática*, pages 137–144, 2007.

[32] José Carlos Bregieiro Ribeiro, Mário Zenha-Rela, and Francisco Fernandez de Vega. An evolutionary approach for performing structural unit-testing on third-party object-oriented java software. In *NICSO 2007: International Workshop on Nature Inspired Cooperative Strategies for Optimization (to appear)*, Studies in Computational Intelligence. Springer-Verlag, 11 2007.

[33] Per Runeson. A survey of unit testing practices. *IEEE Softw.*, 23(4):22–29, 2006.

[34] Ramon Sagarna, Andrea Arcuri, and Xin Yao. Estimation of distribution algorithms for testing object oriented software. In Dipti Srinivasan and Lipo Wang, editors, *2007 IEEE Congress on Evolutionary Computation*, pages –, Singapore, 25-28 September 2007. IEEE Computational Intelligence Society, IEEE Press.

[35] A. Salcianu and M. Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAILTR-949, MIT, May 2004.

[36] Arjan Seesing and Hans-Gerhard Gro. A genetic programming approach to automated test generation for object-oriented software. *ITSSA*, 1(2):127–134, 2006.

[37] Paolo Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, New York, NY, USA, 2004. ACM Press.

[38] Nigel Tracey, John Clark, John McDermid, and Keith Mander. A search-based automated test-data generation framework for safety-critical systems. pages 174–213, 2002.

[39] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong. Establishing structural testing criteria for java bytecode. *Softw. Pract. Exper.*, 36(14):1513–1541, 2006.

[40] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro. Coverage testing of java programs and components. *Sci. Comput. Program.*, 56(1-2):211–230, 2005.

[41] 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.

[42] Stefan Wappler and Frank 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*, pages 1053–1060, New York, NY, USA, 2005. ACM Press.

[43] Stefan Wappler and Joachim 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*, pages 1925–1932, New York, NY, USA, 2006. ACM Press.

[44] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, and S. Katsikas andd K. Karapoulios. Application of genetic algorithms to software testing [application des algorithmes génétiques au test des logiciels]. In *Proceedings of the 5th International Conference on Software Engineering*, pages 625–636, 1992.

[45] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for java programs. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 75–82, New York, NY, USA, 2007. ACM.

Appendix A

Teaching and Investigation Periods

The first biennium of the PhD Program in Information Technologies (*Programa de Doctorado en Tecnologías Informáticas*) consisted on two distinct periods:

- a *teaching period*, with a value of 20 credits, in which the student took a set of courses related with the available research areas;

- a supervised *investigation period*, with a value of 12 credits, in which the student chose a line of investigation and did research in that area.

Table A.1 enumerates the courses taken during the teaching period, and depicts credits and grades obtained in each course.

Table A.2 presents the line of investigation chosen, and the grade obtained at the end of the research period.

The following Sections outline the contents of the courses taken during the teaching period, summarizing the main topics addressed and the novel competences acquired.

A.1 Design and Implementation of Reconfigurable Systems and Parallel Architectures

This course was ministered by professors D. Juan Antonio Gómez Pulido and Miguel Ángel Vega Rodríguez. The main topics addressed were the following:

Course	Credits	Classification
Design and Implementation of Reconfigurable Systems and Parallel Architectures (*Diseño y Síntesis Reconfigurable de Sistemas y Arquitecturas Paralelas*)	8	9 out of 10
Grid Computing and Evolutionary Algorithms (*Computación en Grid y Algoritmos Evolutivos*)	8	9.5 out of 10
Robotics and Artificial Vision (*Robótica y Visión Artificial*)	4	9 out of 10
	Average Classification:	9.2 out of 10

Table A.1: Teaching Period.

Course	Credits	Classification
Techniques for Planning the Improvement of the Efficiency of Evolutionary Algorithms (*Técnicas de Planificación para la Mejora de la Eficiencia de Algoritmos Evolutivos*)	8	9 out of 10

Table A.2: Investigation Period.

Field-Programmable Gate Arrays (FPGAs). FPGAs are semiconductor devices containing programmable logic components and programmable interconnects. Logic blocks can be programmed to perform the function of basic logic gates or more complex combinational functions such as decoders or simple mathematical functions. FPGAs can be employed, for example, for rapid prototyping and for implementing algorithms that can make use of the inherent parallelism offered by their architecture.

Reconfigurable Computing. Reconfigurable computing is a computing paradigm which combines the flexibility of software with the high performance of hardware by making use of flexible high speed computing fabrics – like FPGAs. The main differences, when compared to using ordinary microprocessors or custom hardware, are the ability to make substantial changes to the data path itself (in addition to the control flow) and the possibility of adapting the hardware during runtime by “loading” a new circuit on the reconfigurable fabric.

A.2 Grid Computing and Evolutionary Algorithms

This course was ministered by professors Juan Manuel Sanchez Perez and Francisco Fernándes de Vega.

A State-of-the-art paper on the topic of Evolutionary Testing, entitled "A Primer On Evolutionary Testing", was written as a part of this course.

The main topics addressed were the following:

Algorithms and Optimization. An algorithm is a finite list of well-defined instructions for accomplishing some task that, given an initial state, will proceed through a well-defined series of successive states, eventually terminating in an end-state. Algorithms can be classified according to several criteria, including implementation (parallelism, determinism, ...), design paradigm (greedy, linear, divide and conquer, ...), field of study (search, combinatorial, criptographic, ...) and, most interestingly, complexity – which deals with the amount of time they need to complete compared to their input size. Optimization is, thus, the process of modifying an algorithm to make some aspect of it work more efficiently or use fewer resources.

Grid Computing. Grid computing is a method of computer processing in which different parts of a program are run simultaneously on two or more computers that are communicating with each other over a network; it is a type of segmented or parallel computing, and can be used to enhance the performance of algorithms.

Evolutionary Computation. Evolutionary Computation uses 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. All of these methodologies try to solve problems for which no reasonable fast algorithms have been developed, and they are specially fit for optimization problems.

A.3 Robotics and Artificial Vision

This 8 credit course was ministered by professors Pablo Bustos García de Castro and José Moreno del Pozo ministered.

A report, entitled “Object Recognition from Local Scale-Invariant Features”, was written as a part of this course.

The main topics addressed were the following:

Artificial Vision. As a scientific discipline, Artificial Vision is concerned with the theory and technology for building artificial systems that obtain information from images. The image data can take many forms – such as video sequences, views from multiple cameras, or multi-dimensional data from a medical scanner.

Pattern Recognition and Image Processing. Pattern Recognition is, in general, the act of taking in raw data and taking an action based on the category of the data; however, this analysis can be too complex to be performed without further processing on the data. Feature extraction thus involves simplifying the amount of resources required to describe a large set of data accurately. With Image Processing, in particular, the data is composed by images, and it involves treating those images as a two-dimensional signals and applying standard signal-processing techniques to it. Scale invariant feature transform (SIFT) is one of several computer vision algorithms for extracting distinctive features from images; it can be used for tasks like matching different views of an object or scene and image recognition. The features obtained by SIFT are invariant to image scale, rotation, and partially invariant to changing viewpoints, and change in illumination.

Appendix B

Planning

In this Chapter, the high-level objectives and low-level tasks which will guide this research until the deposit of the PhD thesis in June of 2009 will be discussed. Table B.1 depicts the calendarization for the main goals that will be underlying our studies; it can be thought of as a kind of "mission statement".

Deadline	**Goals**
January 2008	Low-level goals must be clearly defined; partial results with originality, relevance and scientific interest must have been obtained.
June 2008	Relevant results must have been published and discussed in top conferences in the area of research.
December 2008	The main results of the investigation must have been attained; publication must be attempted in relevant international journals with high impact index.
January 2009	The writing of the final thesis must begin.
June 2009	Deposit of the Thesis.
December 2009	Discussion of the Thesis.

Table B.1: High-level goals for the development of the thesis.

In section 6, the main topics of investigation that we will be addressing in following biennium were outlined. Our approach for achieving those objectives includes addressing the research as a phased process, which includes the following stages [28]:

- *State-of-the-art phase* - consists on studying the highest degree of development of a technique at a particular time, which includes testing

different tools, methods of research and data gathering, prototype implementation processes and, mainly, reviewing the existing bibliography on the subject; one must also have the capacity of understanding and employing previously published techniques from different sources, testing ideas, and be familiar with different theories and empirical studies.

- *Prototyping phase* - aims not only to prove the feasibility of the task, but also to discover new problems and challenges that can be (or must be) dealt with.

- *Data Collection phase* - its purpose is that of transforming raw data into insight. During this process, one must take the time to analyze the data generated during the Prototyping tasks from different angles; with any luck, innovative ideas, that can contribute positively to the investigation process, will be found.

- *Writing phase* - its main objective is that of compiling the results of the work developed in order to enable its reviewing by the scientific community. The process of writing the thesis must include publishing papers in relevant conferences or journals, so that concepts and theories can be validated and proved to be relevant.

Table B.2 discriminates both the future tasks and the on-going jobs, and proposes a schedule for performing them.

Task Description	Start	Finish	Duration
Experiments with new Test Objects			
Prototyping	2007/11/15	2008/01/15	2 months
Data Collection	2007/12/15	2008/01/15	1 month
Writing	2007/12/15	2008/01/15	1 month
Method Call Sequence Separation			
Prototyping	2007/12/01	2008/12/15	1 fortnight
Data Collection	2007/12/15	2008/01/15	1 month
Writing	2007/12/15	2008/01/15	1 month
Input Domain Reduction			
State-of-the-art	2008/01/01	2008/02/01	2 months
Prototyping	2008/01/15	2008/03/15	2 months
Data Collection	2008/03/01	2008/05/01	2 months
Writing	2008/04/01	2008/06/01	2 months
Search Domain Sampling			
State-of-the-art	2008/01/01	2008/02/01	2 months
Prototyping	2008/02/01	2008/04/01	2 months
Data Collection	2008/03/01	2008/05/01	2 months
Writing	2008/04/01	2008/06/01	2 months
Assembling a Fully Functional Tool			
State-of-the-art	2008/06/01	2008/06/15	1 fortnight
Prototyping	2008/06/15	2008/09/15	3 months
Data Collection	2008/08/01	2008/09/15	1 month
Writing	2008/08/01	2008/09/01	2 months
Thesis Writing			
Data Collection	2009/01/01	2008/03/01	3 months
Writing	2009/01/01	2008/06/01	6 months

Table B.2: Low-level tasks for the development of the thesis.

Appendix C

Publications

C.1 Using Dynamic Analysis of Java Bytecode for Evolutionary Object-Oriented Unit Testing (SBRC WTF 2007)

Using Dynamic Analysis Of Java Bytecode For Evolutionary Object-Oriented Unit Testing

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

[1]Polytechnic Institute of Leiria (IPL)
Campus 2, Morro do Lena, Alto do Vieiro – Leiria – Portugal

[2]University of Extremadura (UNEX)
C/ Sta Teresa de Jornet, 38 – Mérida – Spain

[3]University of Coimbra (UC)
CISUC, Department of Informatics Engineering, P 3030-290 – Coimbra – Portugal

`jose.ribeiro@estg.ipleiria.pt, fcofdez@unex.es, mzrela@dei.uc.pt`

Abstract. *The focus of this paper is on presenting a methodology for generating and optimizing test data by employing evolutionary search techniques, with basis on the information provided by the analysis and interpretation of Java bytecode and on the dynamic execution of the instrumented test object.*

The main reason to work 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 given test set and to guide the evolutionary search towards reaching specific test goals.

Java bytecode retains enough high-level information about the original source code for an underlying model for program representation to be built. The observations required to select or generate test data are obtained by employing dynamic analysis techniques – i.e. by instrumenting, tracing and analysing Java bytecode.

1. Introduction

Software testing is an expensive process, typically consuming roughly half of the total costs involved in the software development process while adding nothing to the raw functionality of the final product. Yet, it remains the primary method through which confidence in software is achieved. In industry, this process is often done manually – with the responsibility of assessing the quality of a given software product usually falling on the software tester. However, 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 application of evolutionary algorithms to test data generation is often referred to in literature as Evolutionary Testing (Mantere and Alander 2005). 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 input data – test cases – that satisfies a certain test criterion. In 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. The application of search-based strategies for object-oriented unit testing has not yet been investigated comprehensively.

In this paper, we present an approach for guiding the evolutionary search towards generating test sets using coverage metrics derived from the test object's Java bytecode. The main reason to work at the bytecode level is that even when the test object's source code is unavailable, structural testing requirements can still be derived and used to assess the quality of a given test set. The observations required to extract such metric are obtained by employing dynamic analysis techniques – i.e. by instrumenting, tracing and analysing Java bytecode.

In the following section, background on the topics of testing methodologies, quality criteria, evolutionary search techniques and fitness evaluation is provided; related work is reviewed in Section 4. In section 5, we present our methodology for employing dynamic analysis of Java bytecode for test quality assessment and optimization and, on Section 6 the complete framework of our tool is outlined. The concluding chapter resumes the key ideas of this paper and presents some topics for future research.

2. Background

The assessment of the quality of a given test set can be achieved functionally (black-box testing) or structurally (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. Black-box testing is the most widely used testing approach; however, its applicability is often hindered by the need for a formal specification of the test object to be available. With white-box testing, the metrics for measuring the thoroughness of a given test set can be extracted from the structure of the target's source code, or even from compiled code (e.g. Java bytecode).

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; 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 the quality of a given test set and the guidance to the test case selection using white-box criteria generally requires the definition of an underlying model for program representation – usually a control-flow graph (CFG). The CFG is an abstract representation of a given method in a class; control-flow testing criteria can be derived based on such a program representation to provide a theoretical and systematic mechanism to select and assess the quality of a given test set.

Two well known control-flow testing standards to derive testing requirements from the CFG are the all-nodes and all-edges criteria (Vincenzi, Delamaro et al. 2006). The all-nodes criterion requires that each node of a given CFG is executed at least once. To distinguish between instructions that are executed under the normal execution of the

program from others that require an exception to be executed, this criterion can be subdivided into two non-overlapping testing criteria so that the testing activity can focus on different aspects of a program at a time:

- all-nodes-exception-independent (All-Nodes$_{ei}$): requires every node of the CFG reachable through an exception-free path to be executed at least once.
- all-nodes-exception-dependent (All-Nodes$_{ed}$): requires every node of the CFG not reachable through an exception-free path to be executed at least once.

Conversely, the all-edges criterion requires that each control-flow deviation is executed at least once. To consider the control-flow in relation to the exception-handling mechanism, this criterion also is subdivided into two non-overlapping testing criteria: all-edges-exception-independent (All-Edges$_{ei}$) and all-edges-exception-dependent (All-Edges$_{ed}$).

The observations needed to assemble the metrics required by these criteria can be collected by abstracting and modelling the behaviours programs exhibit during execution – either by static or dynamic analysis techniques (Tracey, Clark et al. 2002). 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. In contrast, static analysis involves the construction and analysis of an abstract mathematical model of the system (e.g. symbolic execution). Static analysis 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.

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 (Kinneer, Dwyer et al. 2006). Dynamic monitoring for events in Java software can be achieved through instrumentation of Java bytecode.

Bytecode is an assembly-like language that retains much of the high-level information about the original source program. Class files (i.e. compiled Java programs containing bytecode information) are a portable binary representation that contains class related data such as the class name, its superclass name, information about the variables and constants, and the bytecode instructions of each method (Vincenzi, Maldonado et al. 2005). Using bytecode as the basis for building the CFG allows broadening the scope of applicability of software testing tools, since the target object's source code is often unavailable; it 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 used successfully for the unit testing of procedural software, and their application to the generation of quality test data for object-oriented software is an active field of research. 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 (Wappler and Lammermann 2005):

- At the least, an instance of the class under test is needed.
- Additional objects, which are required (as parameters) for the creation of the object under test and for the invocation of the method under test, must be available. Again, for the creation of these additional objects, more additional objects may be required.
- Depending on the kind of test, the participating objects may have to be put into particular states in order for the test scenario to be processed in the desired way. Consequently, method calls must be issued for these objects.

A fitness function for object-oriented evolutionary testing must evaluate test cases according to their ability to meet a given test goal. If white-box criteria are employed, the CFG and the monitored execution of the test object are used to access the adequateness of test cases – i.e. if the CFG node and/or path defined as the test goal was exercised by the execution of a particular test case over the test object.

In (Wappler and Wegener 2006a) a distance-based fitness function, which expresses how close the execution of a test case over the test object is to reaching the current test goal, was proposed. This closeness is expressed in terms of three distances:

- The Method Call Distance (dMC): expresses how close the test case execution approached the method under test in terms of the number of methods called. In case of a runtime exception, execution of a method call sequence terminates prematurely, meaning that the method under test is not called.
- The Control Node Distance (dCN): expresses how close execution of the test object approached the target CFG node.
- The Local Problem Node Distance (dPN): expresses how far the test object's execution is away from diverging along the branch of the problem node which leads to the test goal.

The metric dMC works at the test case level, and steers the evolutionary search towards producing feasible test cases – i.e. it ensures that a method call sequence of a given test case generates no runtime exceptions that prevent the method under test from being called.

Metrics dCN and dPN, on the other hand, are employed to cover individual test goals on the test object, and are computed with basis on the CFG. In (Wegener, Baresel et al. 2001), four methodologies – which depend on the CFG and the required test purpose – for guiding the evolutionary search toward reaching particular test goals were outlined, and the corresponding fitness functions were described:

- Node-oriented methods: require the attainment of specific nodes in the CFG (e.g. statement test, condition test).
- Path-oriented methods: require the execution of certain paths in the CFG (e.g. path tests).

- Node-path-oriented methods: require the achievement of a specific node and, from this node, the achievement of a specific path through the CFG (e.g. branch test, segment coverage).
- Node-node-oriented methods: aim to execute program paths that cover certain node combinations of the CFG in a pre-determined sequence without specifying a specific path between nodes (e.g. data-flow criteria).

3. Related Work

Interesting review articles on the topic of Evolutionary Testing include that of McMinn (McMinn 2004), who presents a review of meta-heuristic techniques that have been used in software test data generation, namely Hill Climbing, Simulated Annealing and – most interestingly – Evolutionary Algorithms. Namely, the main achievements in automating test data generation in the areas of structural testing, functional testing, and grey-box testing are summarized. In (Mantere and Alander 2005) an in-depth index of the work developed in the area is provided; topics include genetic algorithms applied to coverage testing, test data generation, testing program dynamics, black-box testing and software quality.

Both works pinpoint the state problem (McMinn and Holcombe 2003) as the main issue to be faced by researchers in this field. It 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 not available to external manipulation. The only way to change the values of these variables is through execution of statements that perform assignments to them. In object-oriented software this can occur through the use of variables that are protected from external manipulation using access modifiers.

The first approach to the field of evolutionary testing for object-oriented software was presented in (Tonella 2004); in this work, input sequences were generated using evolutionary algorithms for the white-box testing of classes. Genetic algorithms were the evolutionary approach employed, with potential solutions (test cases) being represented as chromosomes. A source-code representation was 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) – was defined. A population of individuals, representing the test cases, was evolved in order to increase a measure of fitness, accounting for the ability of the test cases to satisfy a coverage criterion of choice – the proportion of all control and call dependences that lead to the given target. New test cases are generated as long as there are targets to be covered or a maximum execution time is 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 – evolutionary algorithms, provided by popular toolboxes, which are independent from the application domain and offer a variety of predefined, probabilistically well-proven evolutionary operators – cannot be applied.

An approach which built upon an Ant Colony Optimization Algorithm was presented by (Liu, Wang et al. 2005). The focus was 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 were employed. To cover those branches enclosed in private/protected methods without violating encapsulation, call chain analysis on class call graphs was introduced.

In (Wappler and Lammermann 2005) an approach for the automatic generation of test programs for object-oriented unit testing was presented, focusing on the usage of Universal Evolutionary Algorithms. An encoding was proposed that represented object-oriented test programs 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 programs could be transformed into test classes according to popular testing frameworks. The suggested encoding, however, did not prevent the generation of individuals which could not be decoded into test programs without errors; their 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 (Wappler and Wegener 2006b) a different approach to the subject was presented. Potential solutions were encoded using a Strongly-Typed Genetic Programming (STGP) methodology, 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 (Wappler and Wegener 2006a). This methodology yielded very encouraging results. For a simple custom-tailored test cluster, the set of generated test cases achieved full (100%) branch coverage: during the search, 11966 test programs were generated and evaluated, and the resulting test set contained 3 test cases; a control run, in which random test cases where produced for comparison purposes, stopped after having evaluated 43233 test programs (in accordance to the specified termination criteria), and the generated test set achieved a coverage of 66%. In a more complex scenario, four classes where tested and full coverage was achieved for all of the test objects.

In the abovementioned approaches, the underlying model for program representation is built with basis on the test object's source-code; moreover, instrumentation of the test object for extracting tracing information is also done 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, however, already studied in different scenarios. In (Cheon, Kim et al. 2005) an attempt to automate unit testing of object-oriented programs is described. A black-box approach for investigating the use of genetic algorithms for test data generation is employed, and program specifications written in JML are used for test

result determination. The JML compiler was extended to make Java bytecode produce test coverage information. In (Muller, Lembeck et al. 2004), the layout of a symbolic Java virtual machine (SJVM), which discovers test cases using a definable structural coverage criterion with basis on static analysis techniques, is described. Java bytecode is executed symbolically, and the decision whether to enter a branch or throw an exception is based on the earlier constraints, a constraint solver and current testing criterion. The SJVM has been implemented in a test tool called GlassTT. This work, however, doesn't address exception-related and method interaction-related criteria, and only procedural software scenarios are described.

4. Dynamic Analysis Of Java Bytecode For Test Quality Optimization

The focus of this paper is on presenting a methodology for generating and optimizing test data by employing evolutionary search techniques, solely with basis on the information provided by the analysis Java bytecode and on the dynamic execution of the instrumented test object's class files.

In this section, the simple test cluster defined in (Wappler and Wegener 2006a) is used for demonstration purposes. We focus on the `Controller.reconfigure(Config)` public method; its source code is reproduced in Figure 1.

```
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 1. Source code for the method `Controller.reconfigure(Config)`

The source code provided by this example was compiled using JDK 1.5. The bytecode instructions of the compiled `Controller.reconfigure(Config)` public method are depicted in Figure 2.

4.1. Bytecode Analysis

In order to understand the details of Java bytecode, a preliminary discussion on how the Java virtual machine (Lindholm and Yellin 1999) works regarding the execution of the bytecode must take place. A JVM is a stack-based machine; each thread has a JVM stack which stores *frames*. A frame is created each time a method is invoked, and consists of an operand stack, an array of local variables, and a reference to the runtime constant pool of the class of the current method (Haggar 2001).

The array *of local variables* contains the parameters of the method and the values of the local variables. The size of the array of local variables is determined at compile time, and is dependent on the number and size of local variables and formal method parameters. The parameters are stored first, beginning at index 0. If the frame is for a constructor or an instance method, the `this` reference is stored at location 0; location 1 contains the first formal parameter, location 2 the second, and so on. For a static method, the first formal method parameter is stored in location 0, the second in location 1, and so on. The *operand stack* is a LIFO stack used to push and pop values;

its size is also determined at compile time. Certain bytecode instructions push values onto the operand stack; others take operands from the stack, manipulate them, and push the result. The operand stack is also used to receive return values from methods.

In Figure 2, The `aload_1` instruction at location 0 pushes the value from the index 1 of local variable table onto the operand stack – i.e. it pushes the parameter `cfg` of the method `Controller.reconfigure(Config cfg)` onto the top of the operand stack (a reference to an object of type `Config`). The `invokevirtual` instruction at location 1 invokes the instance method `Config.getSignalCount()` on the object `cfg` (popped from the top of the operand stack); the value returned by this method is pushed onto the top of the operand stack. The `iconst_5` instruction at location 4 loads the integer value `5` onto the top of the operand stack. At this point, the operand stack contains two values: the integer `5` on top, and the value returned by the `Config.getSignalCount()` on the bottom. The `if_icmple` instruction loads both those values from the operand stack, and compares them: if `5` is lower than or equal to the value returned from the `getSignalCount` method, instruction flow is transferred to instruction 18.

```
cfg.Controller.public_void_reconfigure(cfg.Config_cfg)
_throws_java.lang.Exception
Code(max_stack = 3, max_locals = 2, code_length = 64)
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 2. Java bytecode for the method `Controller.reconfigure(Config)`

This brief analysis helps to support the following conclusions: firstly, the bytecode instructions contain enough high-level information for coverage criteria to be applied at the bytecode level; secondly, it is possible to group some instructions into a smaller set of basic blocks that can ease the representation of the compiled program using a CFG and, consequently, the application of dynamic analysis and structural coverage metrics on the target object.

The purpose of the `aload_1` instruction is, in fact, to prepare the operand stack for the `getSignalCount` method call at location 2. Equally, the `iconst_5` instruction prepares the stack for the `if` instruction on location 5. We group these instructions into two basic blocks: the first pair is grouped into a Call block; the second pair is grouped into a Basic Instruction block of the sub-type "if".

4.2. CFG Definition and Interpretation

In our approach, bytecode instructions are grouped into a set of basic blocks – namely, Basic Instruction blocks and Call Blocks. These blocks cover the core of nodes required to build the CFG graph.

Basic Instruction blocks encompass regular bytecode instructions, including the decision and branching instructions that can influence control flow – namely the sub-types "if", "goto", "jsr", "switch", "return", "ret", "throw", "sumthrow" and "exit". They are represented in the CFG by Basic Instruction nodes. Call blocks represent bytecode instructions that cause control flow to be transferred to another method; they contain the high-level information needed to identify the method being called, and are represented in the CFG by Call nodes. In our example, bytecode instructions are grouped in accordance to Table 1.

Table 1. Mapping between bytecode instructions, basic instruction blocks, basic intruction block subtypes, and node numbers in the CFG depicted in Figure 3.

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	39	Basic	If	11
38	47	Basic	Throw	12
48	55	Call		13
58	63	Basic	Return	15

Additionally, other types of 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.

As mentioned above, Call blocks transfer control flow to the CFG of another method; the method called, in turn, can return normally or with an exception. In order to differentiate these situations, Return nodes are created. They follow Call nodes, and are transversed when the called method returns regularly; if the called method returns with an exception, either the exception is dealt with internally or control flow jumps to an Exit node that causes the method to return with an exception itself.

Exit nodes follow other nodes that can cause the method to return; a different Exit node is created for each return scenario – including "return" and "throws" instructions, and method call instructions that may return an exception.

Entry nodes identify the starting point of the CFG. They simply indicate the method's entry point; there's only one Entry block node per method.

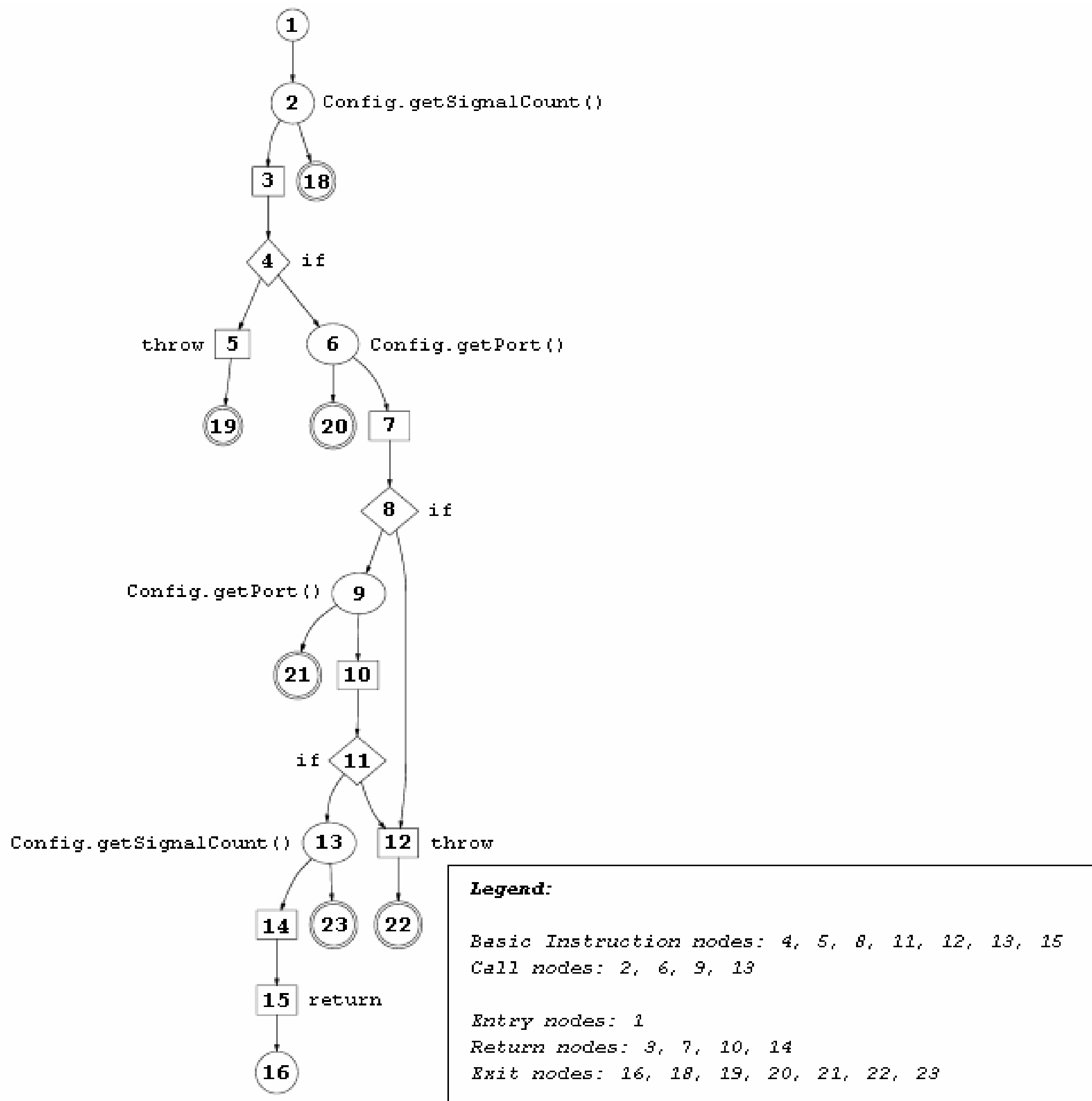


Figure 3. Control-Flow Graph for the method `Controller.reconfigure(Config)`.

The CFG generated for the method `Controller.reconfigure(Config)` is depicted of Figure 3. A brief overview follows: node #1 is the Entry node; it is connected by a directed edge to Call node #2, which transfers control flow to the `Config.getSignalCount()` method; if `Config.getSignalCount()` method returns regularly, Return node #4 is next; if it throws an exception, Exit node #18 is transversed and `Controller.reconfigure(Config)` returns with an exception itself.

4.3. Test Set Evaluation and Optimization

Using the CFG built with basis on the test object's bytecode, it is possible to evaluate thoroughness of the test set with basis on the quality criteria proposed by (Vincenzi, Maldonado et al. 2005; Vincenzi, Delamaro et al. 2006); moreover, it is possible to optimize the test set by employing the evolutionary search techniques proposed by (Wegener, Baresel et al. 2001; Wappler and Wegener 2006a; Wappler and Wegener 2006b). Both methodologies were introduced in the Background section.

In order to access the thoroughness of the testing process according to the all-nodes and all-edges criteria, the test object's class files must be instrumented for Basic block and Call block dispatch, in accordance to the CFG defined as the underlying representation. Dynamic analysis is performed by executing the instrumented test object using each test case of a given test set as input; the trace files produced must then be analyzed for the coverage metrics to be calculated.

```
public void testReconfigure() throws Exception
{
   System.out.println("reconfigure");

   int expected = 8000;

   Config cfg = new Config(9999);
   cfg.setPort(expected);
   Controller instance = new Controller();
   instance.reconfigure(cfg);

   int actual = cfg.getPort();

   assertEquals(expected, actual);
}
```

Figure 4. Sample test case for method `Controller.reconfigure(Config)`

Exception-independent control-flow analysis implies the coverage of the bytecode instructions represented by Basic Instruction nodes and Call nodes (in the case of all-edges$_{ei}$ criterion, by exercising all available branches). Trace files for the execution of the instrumented test object defined in Figure 2 using the test case depicted in Figure 4 yield the transversal of nodes #2, #4, #6, #8, #13 and #15 – i.e. 66% all-nodes$_{ei}$ coverage (nodes #5 and #12 aren't exercised) and 90% all-edges$_{ei}$ coverage (edge #11→#12 isn't exercised; edges beginning at virtual nodes aren't considered) is achieved.

When employing exception-dependant coverage criteria, Exit nodes and Basic Instruction nodes of the subtype "jsr" constitute the focus of the analysis. The all-nodes$_{ei}$ criterion implies the coverage of `catch` and `finally` Java blocks; additionally, the all-edges criterion also implies the transversal of all the edges that lead to Exit nodes that follow Call nodes.

Specific fitness functions have to be defined for each coverage criterion; we employ the methodologies proposed by (Wegener, Baresel et al. 2001). Each individual fitness function, depending on the coverage criterion of choice, is defined as follows (discussion includes examples related to the coverage information extracted from the trace files described above):

- all-nodes$_{ei}$: a node-oriented fitness function is used, which allows the search to be guided towards achieving every individual test goal – e.g. test cases that exercise nodes #5 and #12 must be created.
- all-edges$_{ei}$: a node-path-oriented fitness function is used, which allows the search to be guided towards reaching a specific problem node and, from there, following a certain path – e.g. a test case that considers node #11 as the problem node and transverses edge #11→#12 must be created.
- all-nodes$_{ed}$: a node-path-oriented fitness function is used, which allows the coverage of all bytecode instructions that are encompassed in `catch` and `finally`

blocks, and that can be reached through a `jsr` bytecode function. Basic Instruction nodes of the subtype "jsr" are the problem nodes.

- all-edges$_{ed}$: a node-node-oriented fitness function is used. In addition to the nodes encompassed by `catch` and `finally` blocks, test cases must be generated that reach every single Call node and, from there, transverse the Exit node that corresponds to the called methods exceptional return – e.g. test cases must be generated that consider nodes #18 to #23 as individual goals for the evolutionary search.

5. Framework Description

The focus of our tool is on the creation and optimization of a test set that maximizes code coverage. Optimization occurs at the test set level and at the test case level: we aim to generate a set that can help gain confidence in the software under test using white-box metrics, and to generate the shortest sequence for a given test goal.

The process of CFG building, bytecode instrumentation and event tracing is achieved with the aid of Sofya (Kinneer, Dwyer et al. 2006), a dynamic analysis framework developed at the University of Nebraska, USA, that is particularly suited for developing dynamic analysis tools. 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 files produced by the executions of instrumented programs.

In the context of our tool, Sofya is employed to instrument classes for structural event dispatch. Basic Block instrumentation enables the observations of the virtual Entry, Exit and Return blocks, Call blocks and Basic Instruction blocks transversed during a given program execution. Our tool automatically executes instrumented programs, and compares the trace files produced to the statically generated CFGs in order to compute the fitness function.

For evolving the set of test cases, the Evolutionary Computation in Java (ECJ) package (Luke, Panait et al. 2007) is used in a similar fashion to the one proposed in (Wappler and Wegener 2006a; Wappler and Wegener 2006b). ECJ is a research package, developed at the George Mason's University, USA, that incorporates several Universal Evolutionary Algorithms, and includes built-in support for Set-Based Strongly-Typed Genetic Programming. It is highly flexible, having nearly all classes and their settings being dynamically determined at runtime by user provided parameter files.

Parameter files containing all the constraints defined by the function set are automatically generated by our tool: firstly, the Test Cluster and the Type Set for the Class Under Test are extracted by means of the Java Reflection API; then, the Extended Method Call Dependence Graph (EMCDG) is computed, and a Function Set for each of the public methods is derived; finally, ECJ parameter files are automatically generated for each of the function sets produced.

jUnit is used as the front-end for our tool, as it constitutes both the starting and ending points of the software testing process: the initial population of test cases can optionally be derived from those defined by the user using jUnit (the initial population can also be created automatically), and the generated test programs can be transformed into test classes that can be loaded into the jUnit framework. The major usage scenario is the generation of test cases that complete a test set in order to maximize code coverage. The rationale for minimizing the length of the method call sequence of test cases is that of simplifying the user's task of defining assertions.

6. Conclusions and Future Work

This paper presents the rationale and introduces the methodology for generating and optimizing test sets with basis on metrics derived from the dynamic analysis of the test object's Java bytecode. A Control-Flow Graph is used as the underlying model for program representation, and it is build solely with basis on the high-level information extracted from the Java bytecode of the test object. Bytecode instructions are grouped into a smaller set of Basic Instruction and Call blocks with the intention of easing the representation of the test object's control flow, and additional virtual nodes are defined to facilitate the dynamic analysis phase. The methodology for evaluating the test set includes instrumenting the bytecode for basic block analysis and structural event dispatch, and executing the instrumented test object using the generated test cases as inputs, with the intention of collecting trace files with which to derive coverage metrics. Methodologies for defining fitness functions in order to achieve the particular criteria-related test goals are introduced. A general overview on how our automated software testing tool is integrated is given.

Evolutionary testing is an emerging methodology for automatically generating high quality test data. Future work includes performing a case-study in a real development context in order to demonstrate the usefulness and applicability of the methodology and experiment different approaches to the evolutionary paradigm employed. Namely, we aim to fine-tune the fitness functions employed for working at the bytecode level. Further research must also be made on the topic facilitating the user's task of defining assertions for the generated test cases (e.g. by minimizing the length of method call sequence of test cases) and on the possibility of using distinct strong-typing mechanisms for the definition of the constraint imposed by the object-oriented paradigm.

7. References

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

Haggar, P. (2001, 2001/07/01). "Java bytecode: Understanding bytecode makes you a better programmer " IBM developerWorks Retrieved 2007/04/01, from http://www-128.ibm.com/developerworks/ibm/library/it-haggar_bytecode/.

Kinneer, A., M. Dwyer, et al. (2006). Sofya: A Flexible Framework for Development of Dynamic Program Analyses for Java Software, Department of Computer Science and Engineering, University of Nebraska - Lincoln.

Lindholm, T. and F. Yellin (1999). The Java virtual machine specification. Harlow, Addison-Wesley.

Liu, X., B. Wang, et al. (2005). Evolutionary search in the context of object-oriented programs. MIC2005: The Sixth Metaheuristics International Conference, Vienna, Austria.

Luke, S., L. Panait, et al. (2007). "ECJ 16: A Java evolutionary computation library." from http://www.cs.gmu.edu/~eclab/projects/ecj/.

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

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

McMinn, P. and M. Holcombe (2003). The state problem for evolutionary testing. Genetic and Evolutionary Computation Conference, Chicago, USA, Springer-Verlag.

Muller, R. A., C. Lembeck, et al. (2004). A symbolic Java virtual machine for test case generation. Proceedings of IASTED Conference on Software Engineering: 365-371.

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

Tracey, N., J. Clark, et al. (2002). A search-based automated test-data generation framework for safety-critical systems, Springer-Verlag New York, Inc.

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

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

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

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

Wappler, S. and J. Wegener (2006b). Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation. Seattle, Washington, USA, ACM Press: 1925-1932.

Wegener, J., A. Baresel, et al. (2001). "Evolutionary test environment for automatic structural testing." Information & Software Technology 43(14): 841-854.

C.2 eCrash: a Framework for Performing Evolutionary Testing on Third-Party Java Components (CEDI JAEM 2007)

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 that 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 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 at providing 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 chapter, related work is reviewed. In chapter 3, the framework of our tool is outlined, and a case study that illustrates the methodology is described in chapter 4. The concluding chapter 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

The 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.

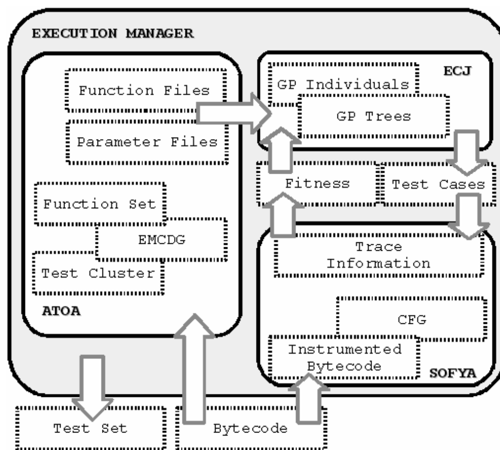


Figure 1. Framework overview

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.

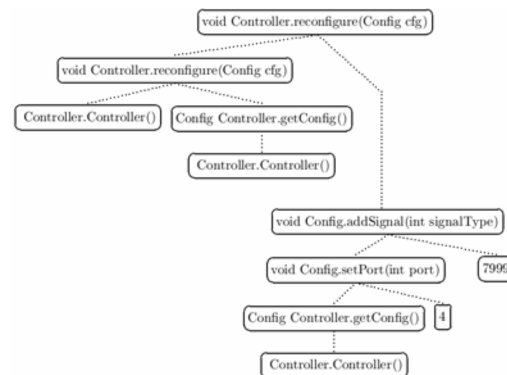


Figure 4. Example GP tree

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.

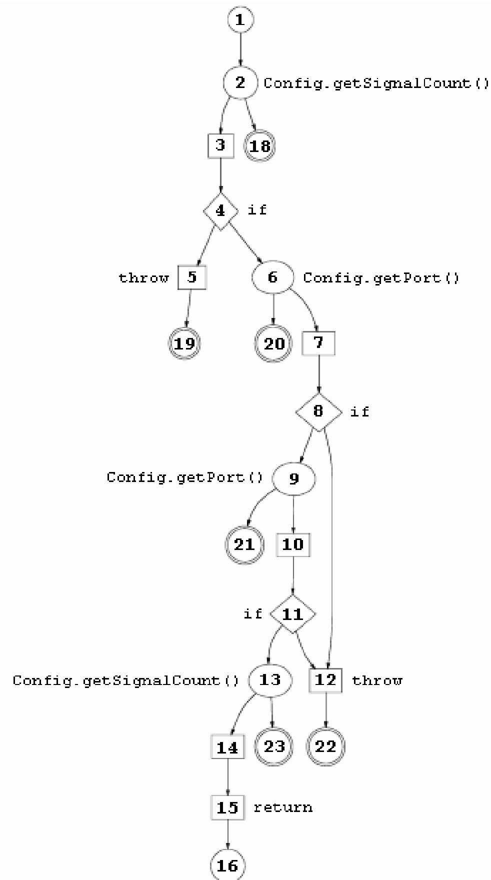
```
package testCases;
import testObject.*;

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 (e.g.: block 1 in Figure 6), Exit blocks (e.g.: 18 to 23), and Return blocks (e.g.: 3, 7, 10, 14). These Virtual blocks encompass no bytecode instructions, and are used to represent certain control flow hypothesis. For this case study's MUT, all the Basic Instruction blocks (4, 5, 8, 11, 12, 15) and 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 full 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.

C.3 An Evolutionary Approach For Performing Structural Unit-Testing On Third-Party Object-Oriented Java Software (NICSO 2007)

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
```

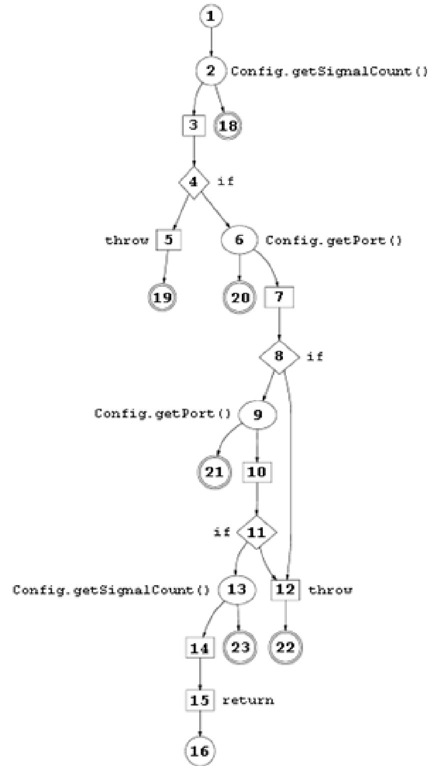


Fig. 4. Bytecode instructions *(left)* and CFG *(right)* for the `Controller.reconfigure(Config)` MUT of the "Controller and Config" test cluster.

`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

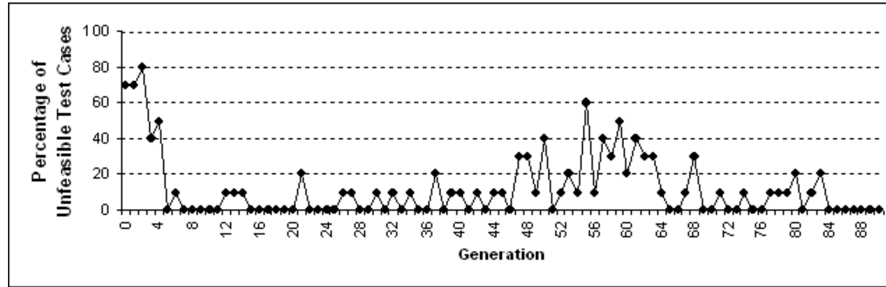


Fig. 5. Percentage of unfeasible test cases per generation for the longest running evolutionary search.

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)

Appendix D

Certification Document – Grades Obtained During the Teaching and Investigation Periods

UNIVERSIDAD DE EXTREMADURA

CERTIFICACION ACADEMICA PERSONAL
ESTUDIOS DE DOCTORADO

DATOS DEL ALUMNO/A:

Nombre y apellidos: JOSÉ CARLOS BREGIEIRO RIBEIRO
DNI: 11962665
Fecha de nacimiento: 10-01-1981
Acceso: TÍTULO SUPERIOR EXTRANJERO (ACCESO DOCTORADO). Universidad: UNIVERSIDAD EXTRANJERA
Título de acceso: ESTUDIOS EXTRANJEROS
Programa de Doctorado: D013 TECNOLOGÍAS INFORMÁTICAS (TIN)
Departamento: INFORMÁTICA
Nº de expediente: 23

Eugénia Maria Lucas Ribeiro
Administradora do
Instituto Politécnico de Leiria

D. CÁNDIDO MURIEL PÉREZ, Jefe de la Sección de Becas y Tercer Ciclo, CERTIFICA que los datos que se reseñan a continuación son exactos, y reflejan el estado actual del expediente académico del alumno /a cuyos datos figuran en la cabecera de este documento.

PERÍODO DE DOCENCIA

CÓDIGO	CURSOS Y/O SEMINARIOS	CRÉD	T	CURSO	CONV	CALIFICACIÓN
313123	Diseño y Síntesis Reconfigurable de Sistemas y Arquitecturas Paralelas	8	F	05-06	JUN	SOBRESALIENTE 9
313124	Computación en Grid y Algoritmos Evolutivos	8	F	05-06	JUN	SOBRESALIENTE 9,5
313129	Robótica y visión artificial	4	F	05-06	JUN	SOBRESALIENTE 9

TIPO	CRÉDITOS SUPERADOS
(F) FUNDAMENTAL	20,0
Total..	20,0

La superación de los créditos señalados acredita la superación del período de docencia de tercer ciclo de estudios universitarios.

CALIFICACIÓN MEDIA: SOBRESALIENTE (9,2)

PERÍODO DE INVESTIGACIÓN

CÓDIGO	CURSOS Y/O SEMINARIOS	CRÉD	T	CURSO	CONV	CALIFICACIÓN
313142	Técnicas de Planificación para la mejora de la eficiencia de Algoritmos Evolutivos	12	R	06-07	JUN	SOBRESALIENTE 9

TIPO	CRÉDITOS SUPERADOS
(R) TRABAJO DE INVESTIGACION	12,0
Total..	12,0

TESIS DOCTORAL leída el día , obteniendo la Calificación: .

OBSERVACIONES:
- Los estudios realizados con anterioridad al curso 1998/99 son regulados por el R.D. 185/85, de 23 de enero (BOE de 16 de febrero).
- Los estudios realizados a partir del curso 1998/99 son regulados por el R.D. 778/98, de 30 de abril (BOE de 1 de mayo).
- Según el art. 3.2 de ambos Reales Decretos, cada crédito equivale a diez (10) horas lectivas.

Para que conste, y a petición del interesado/a, expido esta certificación con el visto bueno del Secretario General y el sello de la Sección, a 11 de octubre de 2007.