Contribuciones a la Mejora de las Técnicas para Test de Software Orientado a Objetos Mediante Programación Genética

Contributions for Improving Genetic Programming-Based Approaches to the Evolutionary Testing of Object-Oriented Software

José Carlos Bregieiro Ribeiro

Doctor of Philosophy Dissertation
Spain, 2010

UNIVERSIDAD DE EXTREMADURA

Thesis for the Degree of Doctor of Philosophy

This Thesis was submitted to the University of Extremadura
in accordance with the criteria necessary for the award
of the Doctorate Degree with European Mention.

Organization:

Departamento de Tecnología de los Computadores y de las Comunicaciones; Universidad de Extremadura; Mérida; España

Title:

Contribuciones a la Mejora de las Técnicas para Test de Software Orientado a Objetos Mediante Programación Genética *(Contributions for Improving Genetic Programming-Based Approaches to the Evolutionary Testing of Object-Oriented Software)*

Author:

José Carlos Bregieiro Ribeiro

Supervisors:

Francisco Fernández de Vega (Universidad de Extremadura, España)
Mário Alberto Zenha-Rela (Universidade de Coimbra, Portugal)

© José Carlos Bregieiro Ribeiro 2010

Dr. D. Francisco Fernández de Vega, profesor titular de la Universidad de Extremadura, España

CERTIFICA:

que la presente memoria, titulada "Contribuciones a la Mejora de las Técnicas para Test de Software Orientado a Objetos Mediante Programación Genética" *("Contributions for Improving Genetic Programming-Based Approaches to the Evolutionary Testing of Object-Oriented Software")* ha sido realizada por D. José Carlos Bregieiro Ribeiro bajo mi dirección en el Departamento de Tecnología de los Computadores y de las Comunicaciones de la Universidad de Extremadura.

Y para que conste, y en cumplimiento de la legislación vigente, firmo la presente.

Dr. D. Francisco Fernández de Vega

Dr. D. Mário Alberto Zenha-Rela, profesor titular de la Universidade de Coimbra, Portugal

CERTIFICA:

que la presente memoria, titulada "Contribuciones a la Mejora de las Técnicas para Test de Software Orientado a Objetos Mediante Programación Genética" *("Contributions for Improving Genetic Programming-Based Approaches to the Evolutionary Testing of Object-Oriented Software")* ha sido realizada por D. José Carlos Bregieiro Ribeiro bajo mi dirección en el Departamento de Tecnología de los Computadores y de las Comunicaciones de la Universidad de Extremadura.

Y para que conste, y en cumplimiento de la legislación vigente, firmo la presente.

Dr. D. Mário Alberto Zenha-Rela

To my parents

Contents

Acknowledgements **vii**

Abstract **ix**

List of Abbreviations **xi**

List of Figures **xiii**

List of Tables **xv**

List of Listings **xix**

1 Introduction **1**
1.1 Major Contributions 2
1.2 Thesis Structure 3

2 Background and Terminology **5**
2.1 Object-Orientation 5
2.1.1 Objects and Classes 8
2.1.2 Encapsulation, Inheritance and Polymorphism 9
2.2 Software Testing 10
2.2.1 Levels of Testing 11
2.2.2 Testing Strategies 14
2.2.3 Static Analysis and Dynamic Analysis 16
2.3 Evolutionary Algorithms 18
2.3.1 Genetic Programming 21
2.4 Evolutionary Testing 26
2.4.1 Object-Oriented Evolutionary Testing 28
2.5 Summary 34

3 Related Work **37**
3.1 Object-Oriented Evolutionary Testing Techniques 37
3.1.1 Genetic Algorithms-based Approaches 38
3.1.2 Genetic Programming-based Approaches 41
3.1.3 Other Metaheuristics-based Approaches 45
3.2 Conclusions 49

4 A Genetic Programming-based Framework for the Evolutionary Testing of Object-Oriented Software **51**
4.1 Methodology Overview 52
4.2 Technical Approach 53
4.3 Test Object Analysis 55
4.3.1 Test Object Instrumentation and Control-Flow Graph Creation 55
4.3.2 Test Cluster Definition 59
4.3.3 Function Set Generation 62
4.3.4 Evolutionary Search Parameterisation 62
4.4 Test Data Generation 64
4.4.1 Setting Up The Evolutionary Run 64
4.4.2 Evolving Test Programs 66
4.4.3 Decoding Test Programs 66
4.4.4 Evaluating Test Programs 67
4.5 Summary 70

5 A Strategy for Evaluating Test Programs for the Evolutionary Testing of Object-Oriented Software **71**
5.1 Control-Flow Graph Nodes' Weights Reevaluation 72
5.2 Evaluation of Feasible Test Cases 73
5.3 Evaluation of Unfeasible Test Cases 73
5.4 Experimental Studies 74
5.4.1 Probabilities of Operators Study 75
5.4.2 Evaluation Parameters Study 76
5.4.3 Discussion 76
5.5 Summary 80

6 Employing Purity Analysis for Reducing the Input Domain of Object-Oriented Evolutionary Testing Problems **81**
6.1 Purity Analysis 82
6.1.1 Related Work 83

6.2 Purified Function Set Generation 85
6.3 Experimental Studies 88
6.3.1 Input Domain Size Study 91
6.3.2 Test Data Generation Results Study 92
6.3.3 Discussion 94
6.4 Summary 95

7 An Adaptive Approach to the Evolutionary Testing of Object-Oriented Software 97
7.1 Adaptive Evolutionary Algorithms 98
7.2 Adaptive Evolutionary Testing Strategy 99
7.2.1 Constraint Selection Ranking Adaptation Strategy . 103
7.3 Experimental Studies 105
7.4 Summary 109

8 Enabling Object Reuse on Genetic Programming-based Approaches to Object-Oriented Evolutionary Testing 111
8.1 The Object Reuse Operator 113
8.1.1 At-Nodes 114
8.1.2 Valid Replaced-Destination Node Pairs 116
8.1.3 Replaced-Destination Node Pair Selection 117
8.1.4 Method Call Tree Linearisation 117
8.2 The Reverse Object-Reuse Operator 118
8.3 Experimental Studies 119
8.3.1 Results and Discussion 120
8.4 Related Work 121
8.5 Summary 125

9 Conclusions and Future Work 127

Bibliography 131

A Publications 147

B Example ECJ Parameter and Function Files 151

C Resumen en Español 155

Acknowledgements

I would like to start by expressing my sincere gratitude to my advisors, Francisco Fernández de Vega and Mário Alberto Zenha-Rela, for giving me the privilege of working with them and for guiding me towards the goal. I owe much of my research to their availability, their support, their enthusiasm.

I also want to thank my colleagues in the Department of Computer Science in the Polytechnic Institute of Leiria, especially those who shared my problems and worries.

A special word goes to all my friends; they were the ones who "made my life happen while I was busy making other plans".

I would also like to acknowledge the help and contributions of all the researchers – even those whom I did not have the chance to meet in person – who have inspired me in this work.

Finally, I would like to take this opportunity for sending a big kiss to Marta, my mum, my dad, and my brother. I simply cannot think of a way to express my gratitude in words. I thank them most of all.

Abstract

Software Testing is the process of exercising an application to detect errors and to verify that it satisfies the specified requirements. It is an expensive process, typically consuming roughly half of the total costs involved in software development; automating Test Data generation is thus vital to advance the state-of-the-art in Software Testing. The application of Evolutionary Algorithms to Test Data generation is often referred to as Evolutionary Testing. The goal of Evolutionary Testing is to find a set of Test Programs which satisfies a particular test criterion. The focus of this work was put on developing a Genetic Programming-based solution for evolving Test Data for the structural Unit Testing of Object-Oriented programs.

The technical approach to Object-Oriented Evolutionary Testing proposed involves representing Test Programs using the Strongly-Typed Genetic Programming paradigm. Test Program quality evaluation includes instrumenting the Test Object, and executing it using the generated Test Programs with the intention of collecting trace information with which to derive coverage metrics. The aim is that of efficiently guiding the search process towards achieving full structural coverage of the program under test.

The foremost objectives of this work were those of defining strategies for addressing the challenges posed by the Object-Oriented paradigm, and of proposing methodologies for enhancing the efficiency and the effectiveness of search-based approaches to Software Testing.

Relevant contributions include:

- the introduction of a novel strategy for Test Program evaluation and search guidance;

- the presentation of an Input Domain Reduction methodology based on the concept of Purity Analysis;

- suggesting an adaptive methodology for promoting the introduction of relevant instructions into the generated Test Programs by means of Mutation; and
- the proposal of an Object Reuse methodology for Genetic Programming-based approaches to Evolutionary Testing, which allows a single object instance to be used as a method parameter multiple times.

The advances attained resulted in the development and implementation of the *eCrash* Test Data generation tool, which embodies the approach to Object-Oriented Evolutionary Testing proposed; special attention was paid to improving the level of automation of both the static Test Object analysis and the iterative Test Data generation processes.

Keywords: Evolutionary Testing, Object-Orientation, Search-Based Software Engineering, Strongly-Typed Genetic Programming, Test Program Evaluation, Input Domain Reduction, Adaptive Evolutionary Algorithms, Object Reuse

List of Abbreviations

ADF Automatically Defined Function 121

API Application Programming Interface 30

ATOA Automatic Test Object Analysis (eCrash module) 54

CCN Cyclomatic Complexity Number 119

CFG Control-Flow Graph 16

CGP Cartesian Genetic Programming 124

CUT Class Under Test 29

EA Evolutionary Algorithm 18

ECJ Evolutionary Computation in Java 55

EMCDG Extended Method Call Dependence Graph 62

GA Genetic Algorithm 20

GP Genetic Programming 21

IP Implicit Parameter (of a method) 87

JDK Java Development Kit 30

JML Java Modelling Language 39

JVM Java Virtual Machine 7

MCS Method Call Sequence 29

MCT Method Call Tree 52

MIO Method Information Object 67

MUT Method Under Test 29

OR Object Reuse 111

PDGP Parallel Distributed Genetic Programming 124

Pn	Explicit Parameter n (of a method)	87
PTC2	Probabilistic Tree Creation 2	102
RE	Return Value (of a method)	87
SBSE	Search-Based Software Engineering	26
SBST	Search-Based Software Testing	48
STGP	Strongly-Typed Genetic Programming	26
TOI	Test Object Instrumentation (eCrash module)	54
TPEM	Test Program Evaluation and Management (eCrash module)	54
TPG	Test Program Generation (eCrash module)	54

List of Figures

2.1 Compiling and running an application using the Java technology. . . . 8
2.2 The Waterfall Model for Software development. 13
2.3 Testing techniques and types of defects. 15
2.4 Control-Flow Graph for the `search` method of the `Stack` class. . . . 17
2.5 Flowchart for the Genetic Programming paradigm. 22
2.6 Example of Genetic Programming subtree Crossover. 24
2.7 Example of Genetic Programming subtree Mutation. 24
2.8 Example interpretation of a Genetic Programming syntax tree. . . 25
2.9 Example Method Call Tree. The Method Under Test is the `search` method of the `Stack` class. 32

3.1 Evacon framework overview. 40
3.2 EvoUnit framework overview. 42
3.3 Breeding pipeline used by EvoTest. 44

4.1 Cross-Functional Diagram of the *eCrash* Framework. 56
4.2 Diagram for the Test Object Analysis process. 57
4.3 Diagram for the Test Data generation process. 65

6.1 EMCDG and purified EMCDG for the `Stack` class. 90
6.2 Purified Function Set for the `Stack` class, and entries excluded from the Purified Function Set 90
6.3 Experimental results for the Input Domain Reduction study: average percentage of CFG nodes remaining per generation. 94

7.1 Example Method Call Tree and corresponding Test Program, built using the Function Set defined in Table 7.1. 103
7.2 Experimental Results for the Adaptive Evolutionary Testing study: average percentage of CFG nodes remaining per generation. . . 109

8.1 Object Reuse and Reverse Object Reuse operators overview. . . 113
8.2 Example Method Call Tree without Object Reuse; and corresponding Method Call Sequence and Test Program. 115
8.3 Example Method Call Tree with Object Reuse; and corresponding Method Call Sequence and Test Program. 116
8.4 Experimental results for the Object Reuse study: Average percentage of CFG nodes remaining, average MCS length, and average percentage of feasible individuals per generation. 123

List of Tables

2.1 Internet-based determination of popular programming paradigms. 6
2.2 Internet-based determination of popular programming languages. 7
2.3 Relative cost to repair defects when found at different stages of Software development. 13

4.1 Default constant set to be included in the Test Cluster for the primitive Java data types. 61

5.1 Experimental results for the Probabilities of Operators study: percentage of runs attaining full structural coverage, and average number of generations required to attain full structural coverage. 77
5.2 Experimental results for the Evaluation Parameters study: percentage of runs attaining full structural coverage, and average number of generations required to attain full structural coverage. 78

6.1 Example Test Cluster for the `Stack` Class. 85
6.2 Parameter Purity Analysis results for the `Stack` class. 86
6.3 Data Types required by the Public Members of the `Stack` class. 87
6.4 Data Types provided by the Public Members of the `Stack` class. 87
6.5 Experimental results for the Input Domain Reduction study: number of EMCDG edges and Function Set entries with and without Parameter Purity Analysis. 92
6.6 Experimental results for the Input Domain Reduction study: average number of generations required to attain full structural coverage, and percentage of runs attaining full structural coverage. 93

7.1 Example Function Set and Type Set. 103
7.2 Experimental Results for the Adaptive Evolutionary Testing study: percentage of runs attaining full structural coverage. 108

8.1 Sources of individuals for the Object Reuse experimental study. 119
8.2 Experimental results for the Object Reuse study: percentage of runs attaining full structual coverage, and average number of individuals evaluated per run. 122

List of Algorithms

4.1 Methodology overview. 54
4.2 Algorithm for Method Call Tree linearisation in the absence of At-Nodes. 68
4.3 Algorithm for Test Program synthesis with basis on the Method Call Sequence. 69

6.1 Input Domain Reduction strategy overview. 85
6.2 Algorithm for EMCDG generation with basis on the Test Cluster. 89
6.3 Algorithm for the generation of the purified EMCDG. 91
6.4 Algorithm for the generation of the purified Function Set with basis on the purified EMCDG. 91

8.1 Algorithm for Method Call Tree linearisation in the presence of At-Nodes. 118

List of Listings

2.1 Source code for the `search` method of the `Stack` class. . . . 17
2.2 Bytecode for the `search` method of the `Stack` class. 17
2.3 Example Method Call Sequence, resulting from the linearisation of the Method Call Tree depicted in Figure 2.9. 32
2.4 Example Test Program, synthesised with basis on the Method Call Sequence depicted in Listing 2.3. 32
3.1 Syntax of chromossomes utilised by eToc. 39
8.1 Programs exemplifying object equality verification in Java. . 112
8.2 Example Test Program employing the Object Pool approach to Object Reuse. 125
B.1 Example ECJ Parameter File for `Stack`'s `search` method. . 151
B.2 Example ECJ Function File for `Stack`'s `search` method. . . 154

Chapter 1

Introduction

Software Testing is expensive, typically consuming roughly half of the total costs involved in software development while adding nothing to the raw functionality of the final product [Bei90]. 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 [TCMM02]. 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.

However, automation in this area has been quite limited [McM04], mainly because the exhaustive enumeration of a program's input is unfeasible for any reasonably-sized program, and random methods are unlikely to exercise "deeper" features of software.

Metaheuristic search techniques, like Evolutionary Algorithms (high-level frameworks which utilise heuristics, inspired by genetics and natural selection, in order to find solutions to combinatorial problems at a reasonable computational cost [BFM97]), are natural candidates to address this problem, since the input space is typically large but well defined, and test goal can usually be expressed as a fitness function [Har07a].

The application of Evolutionary Algorithms to Test Data generation is often referred to as Evolutionary Testing [Ton04] or Search-Based Test Data Generation [McM04]. Evolutionary Testing is an emerging methodology for automatically generating high quality Test Data; approaches have

been proposed that focus on the usage of various metaheuristic strategies, including Genetic Algorithms [Ton04, CKP05, DJAR07, IX07, LRW07, IX08, XTdHS08], Genetic Programming [SG06, See06, WW06a, WW06b, AY07a, WS07, Wap07, GR08], Ant Colony Optimization [LWL05], Memetic Algorithms [AY07b, Arc08], Estimation of Distribution Algorithms [SAY07], or Artificial Immune Systems [LR08]. It is, however, a difficult subject, especially if the aim is to implement an automated solution, viable with a reasonable amount of computational effort, which is adaptable to a wide range of Test Objects.

Significant success has been achieved by applying Evolutionary Algorithms to the automatic generation of Test Data for procedural software [McM04, MA05]. The application of search-based strategies to the Software Testing of Object-Oriented Software is, however, fairly recent [Ton04] and is yet to be investigated comprehensively [Har07b, HMZ09].

The focus of this research was on developing a solution for employing Evolutionary Algorithms to automate the generation of Test Sets for the structural Unit Testing of Object-Oriented programs. Our approach involves representing and evolving Test Programs using the Strongly-Typed Genetic Programming technique [Mon95]. The methodology for evaluating the quality of Test Cases includes instrumenting the program under test, and executing it using the generated Test Cases as inputs with the intention of collecting trace information from which to derive coverage metrics. The aim is that of efficiently guiding the search process towards achieving full structural coverage of the program under test.

1.1 Major Contributions

The foremost objectives of the work supporting this Thesis were those of defining strategies for addressing the challenges posed by the Object-Oriented paradigm and of proposing methodologies for enhancing the efficiency and the effectiveness of search-based approaches to Software Testing. The most relevant contributions achieved were the following:

1. presenting a novel strategy for Test Program evaluation and search guidance. The technique proposed involves allowing unfeasible Test Cases (i.e., those that terminate prematurely due to a runtime exception) to be considered at certain stages of the evolutionary search – namely, once the feasible Test Cases that are being bred cease to be interesting;

2. introducing an Input Domain Reduction methodology, based on the concept of Purity Analysis, which allows the identification and removal of entries that are irrelevant to the search problem because they do not contribute to the definition of relevant test scenarios;

3. proposing an adaptive strategy for enhancing Genetic Programming-based approaches to automatic Test Data generation. Adaptive Evolutionary Algorithms are distinguished by their dynamic manipulation of selected parameters during the course of evolving a problem solution; the main contribution of this study is that of proposing an Adaptive Evolutionary Testing methodology for promoting the introduction of relevant instructions into the generated Test Cases by means of Mutation;

4. defining an Object Reuse methodology for Genetic Programming-based approaches to Evolutionary Testing. Object Reuse means that one instance can be passed to multiple methods as an argument, or multiple times to the same method as arguments; in the context of Object-Oriented Evolutionary Testing, it enables the generation of Test Programs that exercise structures of the software under test that would not be reachable otherwise; and

5. presenting the *eCrash* Test Data generation tool for Object-Oriented Software. The *eCrash* tool embodies the approach to Evolutionary Testing presented in this Thesis; improving the level of performance and automation of the Software Testing process was a major concern underlying its development and implementation. Special attention was put on keeping the interference of the tool's users on the Test Object analysis to a minimum, and on mitigating the impact of their decisions in the Test Data generation process.

1.2 Thesis Structure

Chapter 2 starts by providing background information on the subjects addressed throughout the remaining of this document, and Chapter 3 reviews and contextualises related work in the area of Evolutionary Testing. In Chapter 4, the Evolutionary Testing methodology proposed is overviewed, and the *eCrash* automated Test Data generation framework for Object-Oriented Java software is presented.

In Chapters 5 to 8, this work's most significant contributions to the area of Object-Oriented Evolutionary Testing are presented: Chapter 5 describes a novel Test Program Evaluation and search guidance methodology; Chapter 6 details the Input Domain Reduction technique; Chapter 7 explains the adaptive Evolutionary Testing strategy; and Chapter 8 depicts the Object Reuse methodology proposed.

The concluding Chapter presents some final considerations and sets ground for future work.

Chapter 2

Background and Terminology

In Object-Oriented Evolutionary Testing, metaheuristic search techniques are employed to select or generate Test Data for Object-Oriented software. This Chapter provides background information on the most relevant aspects related with this interdisciplinary area; special attention is paid to the subjects of particular interest to our technical approach, which will be presented and described in Chapter 4.

This Chapter is organised as follows. In the following Section, the Object-Oriented paradigm is overviewed; Section 2.2 reviews key Software Testing concepts; Evolutionary Algorithms, and the Genetic Programming technique in particular, are explored in Section 2.3; and Section 2.4 introduces the reader to the Evolutionary Testing area. Finally, in Section 2.5, relevant concepts and terminology are summarised.

2.1 Object-Orientation

The use of Object-Oriented technology is not restricted to any particular language; rather, it applies to a wide spectrum of programming languages, such as C++ [ES90], Java [Sun03], C# [Wil02] and Visual Basic [Bal02]. In [BME+07], Booch *et. al* argue that a language is considered Object-Based if it directly supports data abstraction and classes; an Object-Oriented language is one that is Object-Based but also provides support for encapsulation, inheritance, and polymorphism.

In its basic definition, an object is an entity that contains both data and behavior. This is the key difference between Object-Oriented and Procedu-

Category	Ratings Nov 2009	Delta Nov 2008
Object-Oriented Languages	54.40%	-3.20%
Procedural Languages	41.60%	2.80%
Functional Languages	2.80%	0.20%
Logical Languages	1.30%	0.10%

Table 2.1: Internet-Based determination of popular programming paradigms. Adapted from the TCP-Index website [Ind09] in November 2009.

ral programming methodologies: in Object-Oriented design, the attributes and behavior are contained within a single object, whereas in Procedural (or structured) design the attributes and behavior are normally separated, with data being placed into totally distinct functions or procedures [Wei08]. With the Procedural paradigm, procedures ideally become “black boxes”, where inputs go in and outputs come out. Also, the data is sometimes global, so it is easy to modify data that is outside the scope of the code; this means that access to data is uncontrolled and unpredictable (because multiple functions may have access to the global data).

Object Orientation is currently the most popular programming paradigm (Table 2.1). The fundamental ideas of classes and objects first appeared in Simula [BDMN79], a language for describing systems and for developing simulations. Simula was first presented in the 1960’s, and introduced the concepts of encapsulation and inheritance. In the early 1970’s research led to the presentation of Smalltalk [fITS98]. Smalltalk is a “pure” Object-Oriented programming language in the sense that the Object-Oriented paradigm is enforced: everything is an object (conversely, in C++ it is possible to use non-objects, and in Java the primitive data types are not implemented as objects). The unification of Object-Oriented concepts with the C programming language [KR88] lead to the C++ programming language [ES90] in the 1980’s. C++ is largely a superset of C, in that it provides type checking, overloaded functions, and – most importantly – adds Object-Oriented programming features to C.

In the 1990’s, a group led by James Gosling at Sun Microsystems* developed and released the Java platform. Java is more than just a programming language; like Smalltalk, it is just as much a runtime environment as it is a language.

Java has been one of the most popular – if not the most popular – pro-

*http://www.sun.com/

Programming Language	**Position**			
	Nov-09	*Nov-05*	*Nov-99*	*Nov-84*
Java	1	1	3	-
C	2	2	1	1
PHP	3	4	25	-
C++	4	3	2	10
(Visual) Basic	5	5	5	4
C#	6	7	23	-
Python	7	8	20	-
Perl	8	6	4	-
JavaScript	9	9	17	-
Ruby	10	24	-	-

Table 2.2: Internet-Based Determination of Popular Programming Languages Throughout Time. Adapted from the TCP-Index website [Ind09] in November 2009.

gramming languages of the last decade (Table 2.2). The growth has come not only as a result of the evolution of the language – which made it a perfect match for the Internet because of its portability and its rich set of standard functionalities [BME+07] – but also, and perhaps more significantly, as a consequence of the implementation of its related technologies such as Enterprise JavaBeans [BMH06], Java Server Pages (JSP) [Ber02], and Java 2 Micro Edition (J2ME) [Top02].

Like Smalltalk (but unlike C++), Java includes a rich class library [Sun03] that can be extended. The Java language syntax resembles C++ intentionally, but omits many of its features, such as multiple inheritance, operator overloading, the use of pointers, and C++'s memory management scheme. As a platform-independent environment, Java programs can be somewhat slower than those written in native code. However, advances in compiler and Virtual Machine technologies are bringing performance close to that of native code without threatening portability.

In the Java programming language, all source code is first written in plain text files ending with the `.java` extension. Those source files are then compiled into `.class` files by the `javac` compiler (Figure 2.1). A `.class` file does not contain code that is native to a specific processor; it instead contains Bytecode – the machine language of the Java Virtual Machine (JVM) [LY99]. The `java` launcher tool then runs the application with an instance of the JVM. Because the JVM is available on many different operating systems, the same `.class` files are capable of running on, for

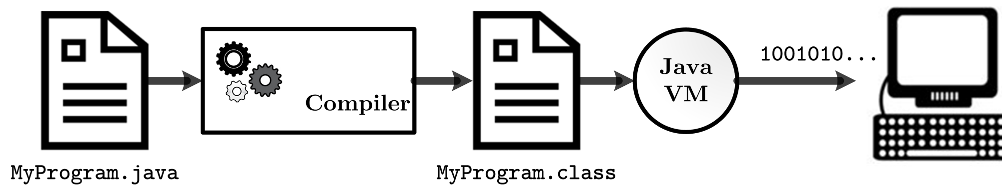

Figure 2.1: Compiling and running an application using the Java technology. Adapted from [ZHR+06].

example, Microsoft Windows, Linux, or Mac Operating System.

Java Bytecode is an assembly-like language that retains much of the high-level information about the original source program [VDMW06]. 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 (e.g., Listing 2.1 on page 17).

2.1.1 Objects and Classes

An object is a software bundle of related state and behavior, which is often used to model the real-world objects. This relation comes from the fact that software objects – much like real world objects – possess state, behaviour and identity [BME+07]. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages); methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Also, each object can be uniquely distinguished from every other object – i.e., each object has a unique address in memory.

Bundling code into individual software objects provides a number of benefits [ZHR+06], including:

- *modularity*, as the source code for an object can be written and maintained independently of the source code for other objects;
- *information-hiding*, as by interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world;
- *code re-use*, because if an object already exists it can be (re-)used again in another program;

- *pluggability and debugging ease*, since if a particular object turns out to be problematic, it can simply be removed from an application, and a different object can plugged as its replacement.

A class is the "blueprint" or "prototype" from which objects are created. A class describes a set of objects that have identical characteristics (data elements) and behaviour (functionality) [Eck02]; in Object-Oriented terms, an object is said to be an instance of a class. A class definition typically includes: fields, which have a type and a value; methods, which have return values, parameters, and may throw exceptions; and constructors, which create new instances of an object for a given class (and have no return values).

In Java, for example, every object is either a reference or primitive type [GJSB05]. Reference types all inherit from the `java.lang.Object` class. Classes, enumerations, arrays, and interfaces are all reference types; examples of reference types include `java.lang.String`, all of the wrapper classes for primitive types such as `java.lang.Double`, and the interface `java.io.Serializable`. There is a fixed set of primitive types: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`.

2.1.2 Encapsulation, Inheritance and Polymorphism

Hiding internal state and requiring all interaction to be performed through an object's methods is known as data Encapsulation – a fundamental principle of Object-Oriented programming. The most important reason underlying the usage of Encapsulation is that of separating the interface from the implementation [Eck02]; this allows establishing boundaries within a data type and hiding its internal mechanism, and prevents client programmers from accidentally treating the internals of an object as part of the interface that they should be using.

Object-Oriented programming allows classes to inherit commonly used state and behavior from other classes. Inheritance expresses this similarity between classes by using the concept of base classes and derived classes: a base class contains all of the characteristics and behaviours that are shared among the classes derived from it. Semantically, inheritance denotes an "is a" relationship; inheritance thus implies a generalization/specialization hierarchy, wherein a subclass specialises the more general structure or behavior of its superclasses.

There is a healthy tension among the principles of encapsulation and inheritance [BME+07]. Encapsulation attempts to provide an opaque barrier behind which methods and state are hidden; inheritance requires opening this interface to some extent and may allow state as well as methods to be accessed [Lis87]. Distinct programming languages trade off support for encapsulation and inheritance in different ways. Java, for example, offers great flexibility, allowing the definition of private members that are accessible only to the class itself, protected members that are accessible to the class and its subclasses (and also to other classes belonging to the same package), and public members which are accessible to all classes.

Polymorphism means "different forms"; it represents a concept in type theory in which a single name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass. Any object denoted by this name is therefore able to respond to some common set of operations [Tho89]; distinction is expressed through differences in behavior of the methods that can be called through the base class. Polymorphism is a condition that exists when the features of dynamic typing and inheritance interact; although inheritance without polymorphism is possible, it is certainly not very useful.

2.2 Software Testing

Software Testing is the process of exercising an application to detect errors and to verify that it satisfies the specified requirements [LW04]. The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances [Mar94]. Despite advances in formal methods and verification techniques, a program 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.

A report [Tas02] issued by the National Institute of Standards and Technology in 2002† stated that software "bugs", or errors, are so prevalent and so detrimental that they cost the United States of America's economy an estimated 59.5 billion dollars annually (about 0.6% of the gross domestic product), with half the costs being supported by software users and the remainder by software developers/vendors. This study also concluded that, even though errors cannot be completely removed, more than a third of these costs could be eliminated by an improved testing infrastructure that

†http://www.nist.gov/public_affairs/releases/n02-10.htm

enables earlier and more effective identification and removal of software defects; these are the savings associated with finding an increased percentage of errors closer to the development stages in which they are introduced. Recent reports[‡] also claim that testing services will grow at a compound annual growth rate of 9.5% from 2008 to 2013 (faster than most other information technology services), and are projected to reach 56 billion dollars by 2013 despite taking a hit from the global economic crisis.

"Test early, test often" is the mantra of experienced programmers; however, developing conformance testing code can be more time consuming and expensive than developing the standard or product that will be tested [Tas02]. Testing is usually regarded as a monotonous and repetitive task, which does not have a predictable end, and may or may not reveal a defect; it involves writing code for drivers and stubs which often add up much more code than the program being tested, and needs to be repeated whenever an enhancement or change is made to the existing code. In the absence of any reasonable automation, this is an activity few programmers (if any) enjoy, and hence is seldom performed sufficiently well [Raj04]. Automating the testing process is thus key to improve the quality of complex software systems that are becoming the norm of modern society [Ber07].

2.2.1 Levels of Testing

Software Testing is a broad term encompassing a wide spectrum of different activities, from the testing of a small piece of code by the developer to the customer validation of a large information system, to the monitoring at run-time of a network-centric service-oriented application [Ber07]. Although testing is involved in every stage of the software life cycle, the testing done at each level of software development is different in terms of its nature and objectives [LW04], and normally targets specific types of faults.

The Encyclopedia of Software Engineering [Mar94] describes four major levels at which testing is conducted: Unit Testing, Integration Testing, System Testing and Acceptance Testing.

- *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, and is often performed in the scope of *Regression Testing* in order to certify that

[‡] http://www.physorg.com/news155992141.html

code modification, bug correction, and any post-production activities have not introduced any additional bugs into previously tested code.

- *Integration Testing* is used to evaluate proper functioning of the integrated modules 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.

- *Acceptance Testing* is done when the completed system is handed over from the developers to the customers or users; its purpose is to give confidence that the system is working rather than to find errors.

Specialised Software Testing stages occur less frequently than general Software Testing stages and are most common for software with well-specified criteria. More specialised testing levels include Usability Testing, Stress Testing, and Performance Testing [LW04].

The historic approach to the software development process, which focuses on system specification and construction, is often based on the Waterfall Model [Roy87]. Although considered flawed [LB03], a recent survey ascertained the popularity of this software development process, with a majority of 35% of managerial and advanced technical respondents indicating that they use it [NL03]. Figure 2.2 shows how the Waterfall Model separates software development into distinct phases with minimal feedback loops.

Testing is inherent to every phase of the Waterfall Model. However, the relative cost of repairing defects increases greatly depending on the stage of software development in which the defect is found; this is due to the re-engineering process that must take place in order to fix the error, which includes unravelling and rewriting the software written to date. Table 2.3 schematises the relation between the stage of software development in which a defect is found and its repair cost; for example, repairing a defect is estimated to cost six times more if done in the post-product release phase instead of in the coding stage.

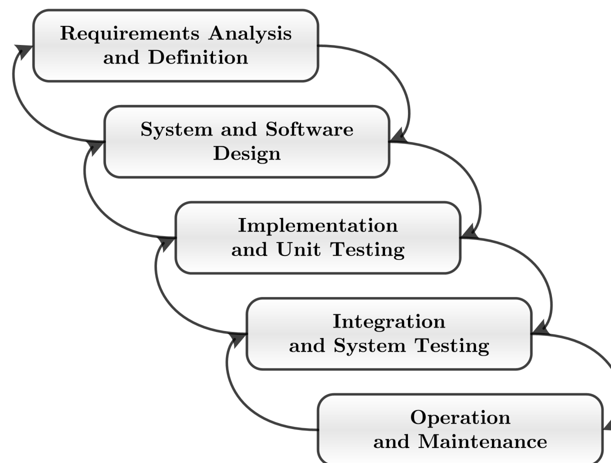

Figure 2.2: The Waterfall Model for Software Development. Adapted from [Tas02].

Requirements Gathering / Architectural Design	Coding / Unit Test	Integration and System Test	Early Costumer Feedback / Beta Test	Post-Product Release
1X	5X	10X	15X	30X

Table 2.3: Relative cost to repair defects when found at different stages of Software development. X is a normalised unit of cost and can be expressed terms of person-hours, money, etc.. Adapted from [Tas02].

Unit Testing

Modern software products typically contain millions of lines of code; precisely locating the source of errors can thus be very resource consuming. Most errors are introduced at the unit stage [Tas02]; Unit Testing is thus a key phase in projects that demand high quality and reliability, and plays a major role in the total testing efforts.

A unit is the smallest testable piece of software – i.e., the smallest component that can be compiled or assembled, linked, loaded, and put under the control of a test harness or driver [Bei90]. Unit Testing is the process of testing the individual subprograms, subroutines, procedures or methods in a program; it typically is performed by executing the unit (i.e., the Test Object) in different scenarios, using a set of 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 the criterion.

The objective of Unit Testing typically involves assessing if a unit satisfies its functional specification, or that its implemented structure matches

the intended design structure; and its primary aim is to uncover errors within a unit, or to gain confidence in its correctness if no errors can be found [WW06b].

Some of the benefits of Unit Testing are:

- testing parts of a project in isolation, without the need to wait for the other parts to be available;
- achieving parallelism in testing, by allowing many developers to test and fix problems simultaneously;
- simplifying debugging, by limiting its scope to a small unit in which to search for errors;
- testing internal conditions (e.g., exception conditions), that may not be easily reached by external inputs to the system as a whole;
- detecting and removing defects at a much smaller cost, in comparison to other (latter) stages of testing in the software development process.

In recent years, Unit Testing has become a much more structured process, mostly due to the availability and dissemination of high quality Unit Testing tools such as JUnit[§] for Java and NUnit[¶] for the .NET language. Fully automating the Unit Test generation process, however, is still an open problem.

2.2.2 Testing Strategies

To gain sufficient confidence that most faults are detected, testing should ideally be exhaustive; since in practice this is not possible [ABHPW09], testers resort to test models and coverage/adequacy criteria to define systematic and effective test strategies that are fault revealing.

Distinct test strategies include Functional (or Black-Box) Testing, and Structural (or White-Box) Testing [Bei90].

- *Functional (or Black-Box) Testing* emphasises on the external behavior of the software entity; it is concerned with showing the consistency between the implementation and its requirements or functional specification. Typically, it does not require knowledge about the internal

§ http://www.junit.org/

¶ http://www.nunit.org/

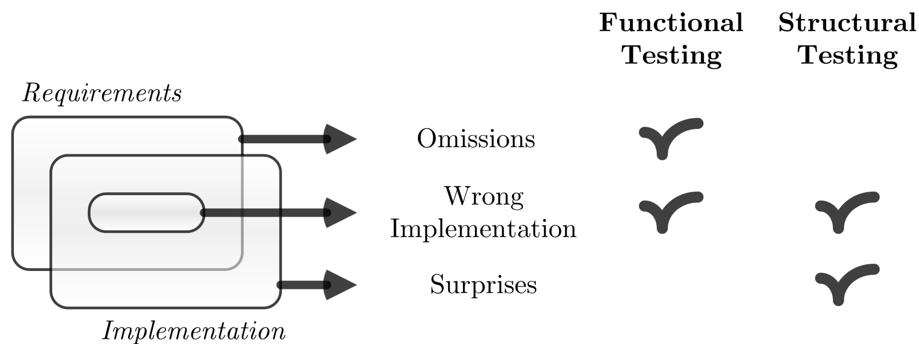

Figure 2.3: Testing techniques and types of defects.

structure of the software under test, and is rather based on input-output relationships: inputs are fed to the Test Object and outputs are observed to determine success or failure of Test Cases.

- *Structural (or White-Box) Testing* focuses on the internal structure of the software entity; Test Case design is performed with basis on the program structure. White-box testing strategies aim at generating Test Cases that cover structural properties of the software under test (e.g., statements, branches, or conditions).

Not all techniques will pick up all types of defects [Raj04]; using Functional Testing alone as a basis for testing will not reveal any "rogue" code (i.e., code that is not found in the requirements), and Structural Testing will not reveal requirements missed in the implementation (Figure 2.3).

Structural Testing

Unit Testing in companies is largely White-box oriented [Run06], mainly because:

- Structural Testing becomes less feasible as the size of the Test Object increases (e.g., when testing entire, large programs, in subsequent testing processes); and
- subsequent testing processes are oriented toward finding different types of errors, not necessarily associated with the program's internal logic (e.g., failure to meet the users' requirements) [MS04].

When Structural Testing is performed, the metrics for measuring the thoroughness of a Test Set are extracted from the structure of the target

object. Traditional structural 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. Statement coverage is widely accepted as the minimum mandatory Structural Unit Testing requirement [Bei90] (e.g., by the IEEE Unit Test standard [IEE87] and by IBM [Hir67]).

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 [VDMW06]. Each node in the graph represents a statement block (i.e., a straight-line piece of code); directed edges are used to represent jumps in the control flow.

The CFG can be extracted from a target object's source code, or even from compiled code; Java Bytecode instructions, for example, 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.

Figure 2.4 depicts an example CFG, representing the `search` method (Listing 2.1) of the `Stack`[‖] Java class, generated with basis on the method's Bytecode (Listing 2.2). Full statement coverage of the `search` method is achieved by a Test Set that traverses all the nodes of the corresponding CFG.

2.2.3 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 [Ern03].

- *Static analysis* involves the construction and analysis of an abstract mathematical model of the system; 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 inspections, program analysis, symbolic analysis and model checking.

[‖] http://java.sun.com/j2se/1.4.2/docs/api/java/util/Stack.html

```java
public synchronized int search(Object o) {
    int i = this.lastIndexOf(o);
    if (i >= 0) {
        return this.size() - i;
    }
    return -1;
}
```

Listing 2.1: Source code for the `search` method of the `Stack` class.

```
public synchronized int search(Object o)

max stack = 2,
max locals = 3,
code length = 19

0: aload_0
1: aload_1
2: invokevirtual lastIndexOf(Object)
5: istore_2
6: iload_2
7: iflt #17
10:aload_0
11:invokevirtual size()
14:iload_2
15:isub
16:ireturn
17:iconst_m1
18:ireturn

Local Variables:
start_pc=0,len=19,ind=0:Stack this
start_pc=0,len=19,ind=1:Object o
start_pc=6,len=13,ind=2:int i
```

Listing 2.2: Bytecode for the `search` method of the `Stack` class.

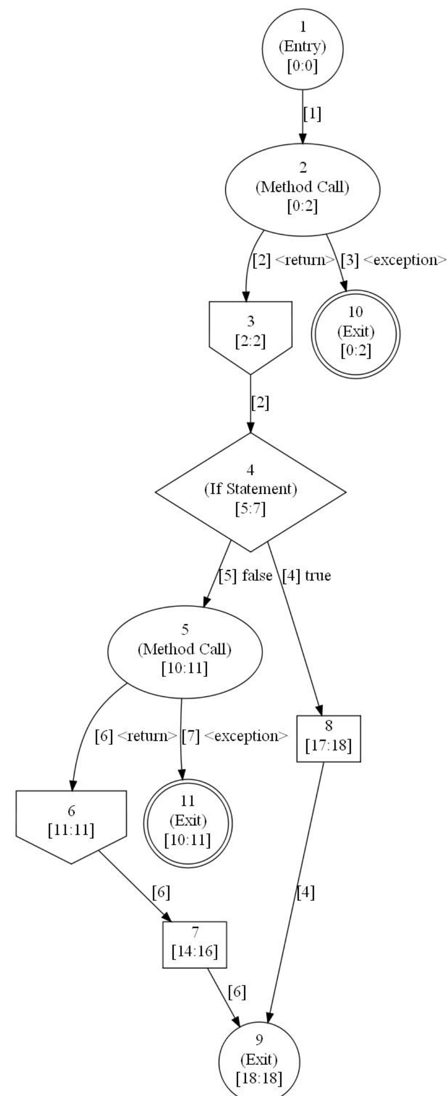

Figure 2.4: CFG for the `search` method of the `Stack` class.

- *Dynamic analysis*, in contrast, involves executing the actual Test Object and monitoring its behaviour; 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.

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.

Dynamic monitoring of structural entities can be achieved by *instrumenting* the Test Object (i.e, adding code probes into a method, which do not alter its state or behaviour, for the purpose of gathering data to be utilised by tracing tools), and *tracing* the structural entities traversed (i.e, logging information about a program's execution) [KDR06]. In Java software, this operation can be effectively performed at the Bytecode level.

2.3 Evolutionary Algorithms

Computing optimal solutions for many problems of industrial and scientific importance is often difficult and sometimes impossible; automating the Test Data generation process is a paradigmatic example. Unlike exact methods, metaheuristics allow solving hard and complex problem instances by delivering satisfactory solutions in a reasonable time.

Metaheuristic search methods can be defined as upper level general templates that can be used as guiding strategies in designing underlying heuristics to solve specific optimization problems. There are many metaheuristic methodologies, and various classification criteria exist, such as [Tal09]:

- *Nature Inspired vs. Non-nature Inspired* – Examples of metaheuristics inspired by natural processes include: Evolutionary Algorithms (EAs) and Artificial Immune Systems [dCT03] from Biology; Particle Swarm Optimization [KL86] from Social Sciences; and Simulated Annealing [LA87] from Physics.

- *Deterministic vs. Stochastic* – A deterministic metaheuristic solves an optimization problem by making deterministic decisions (e.g., Local

Search, Tabu Search [Glo89]); conversely, in stochastic metaheuristics, some random rules are applied during the search (e.g., Simulated Annealing, EAs).

- *Population-based Search vs. Single Solution-based Search*: Single solution-based algorithms (e.g., Local Search, Simulated Annealing) manipulate and transform a single solution during the search; in population-based algorithms (e.g., Particle Swarm Optimization, EAs) a whole population of solutions is evolved.

All metaheuristic search methods try to solve problems for which no reasonable fast algorithms have been developed, and they are especially fit for optimization problems [dV01]. However, the main point of interest in the domain of optimization must not be the design of the best algorithm for all problems – but rather the search for the most adapted algorithm to a given class of problems and/or instances. In fact, the "No Free Lunch Theorem" states that the averaged performance of all search algorithms over all problems is equal [WM97]. That is, if algorithm A performs better than B for a given problem, there is always another problem where B performs better than A. The idea is therefore to use the right algorithm for the right problem.

In designing a metaheuristic two contradictory criteria must be taken into account: exploration of the search space (diversification) and exploitation of the best solutions found (intensification). In intensification, the promising regions are explored more thoroughly in the hope to find better solutions; in diversification, nonexplored regions must be visited to be sure that all regions of the search space are evenly explored and that the search is not confined to only a reduced number of regions. In general, basic single-solution based metaheuristics are more exploitation oriented, whereas basic population-based metaheuristics are more exploration oriented.

Evolutionary Algorithms are the most studied population-based metaheuristics; they are stochastic algorithms, which use simulated evolution as a search strategy to iteratively evolve candidate solutions, using operators inspired by genetics and natural selection. They draw their inspiration from the works of Mendel on heredity [BM09] and from Darwin's studies on the evolution of species [Dar95]. The best known algorithms in this class include Evolution Strategies [Rec65, BS02], Evolutionary Programming [Fog62, Fog99], Genetic Algorithms [Hol62, Gol89] and Genetic Programming [Koz92]. Each of these constitutes a different approach; however,

independently of its class, any Evolutionary Algorithm should possess the following attributes [Mic94]:

- 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 algorithm uses (e.g., population, size, probabilities of applying genetic operators).

Genetic Algorithms (GAs) are the most well known form of Evolutionary Algorithms, having been conceived by John Holland during the late 60’s and early 70’s. 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 GA is called the *genotype*, with the decoded structure being known as the *phenotype*.

Like other Evolutionary Algorithms, GAs are based on the notion of competition. They 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 GA breeding *operators* include Reproduction, Crossover, and Mutation [Hol92]:

- *Reproduction* is the process of copying individuals. They are chosen according to their fitness value;
- *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.

The most commonly employed method for selecting individuals for breeding is Tournament Selection [PLM08]: a number of individuals are chosen at random from the population; these are compared with each other and the best of them is chosen to be the parent. Other common selection methods include Fitness-Proportionate Selection, Linear Ranking, and the Roulette Wheel method [Rut08].

2.3.1 Genetic Programming

Genetic Programming (GP) is specialization of GAs usually associated with the evolution of tree structures; it focuses on automatically creating computer programs by means of evolution [Koz92]. Figure 2.5 provides a flowchart for the GP paradigm.

In order to optimise a computer program, the notion of suboptimal programs – rather than programs which are simply right or wrong – must be allowed [Luk09b]; GP is thus generally interested in the space where there are many possible programs, but it is not clear which ones outperform the others and to what degree.

In most GP approaches, the programs are represented using variable-sized tree genomes. The leaf nodes are called *terminals*, whereas the non-leaf nodes are called *non-terminals* (or functions). Terminals can be inputs to the program, constants or functions with no arguments; non-terminals are functions taking at least one argument. The definition of the terminals and non-terminals depends on the target application.

The *Function Set* is the set of functions from which the GP system can choose when constructing trees. GP builds new trees by repeatedly selecting nodes from a Function Set and putting them together.

The individuals in the initial population are typically randomly generated. There are various distinct approaches to perform this task; two of the

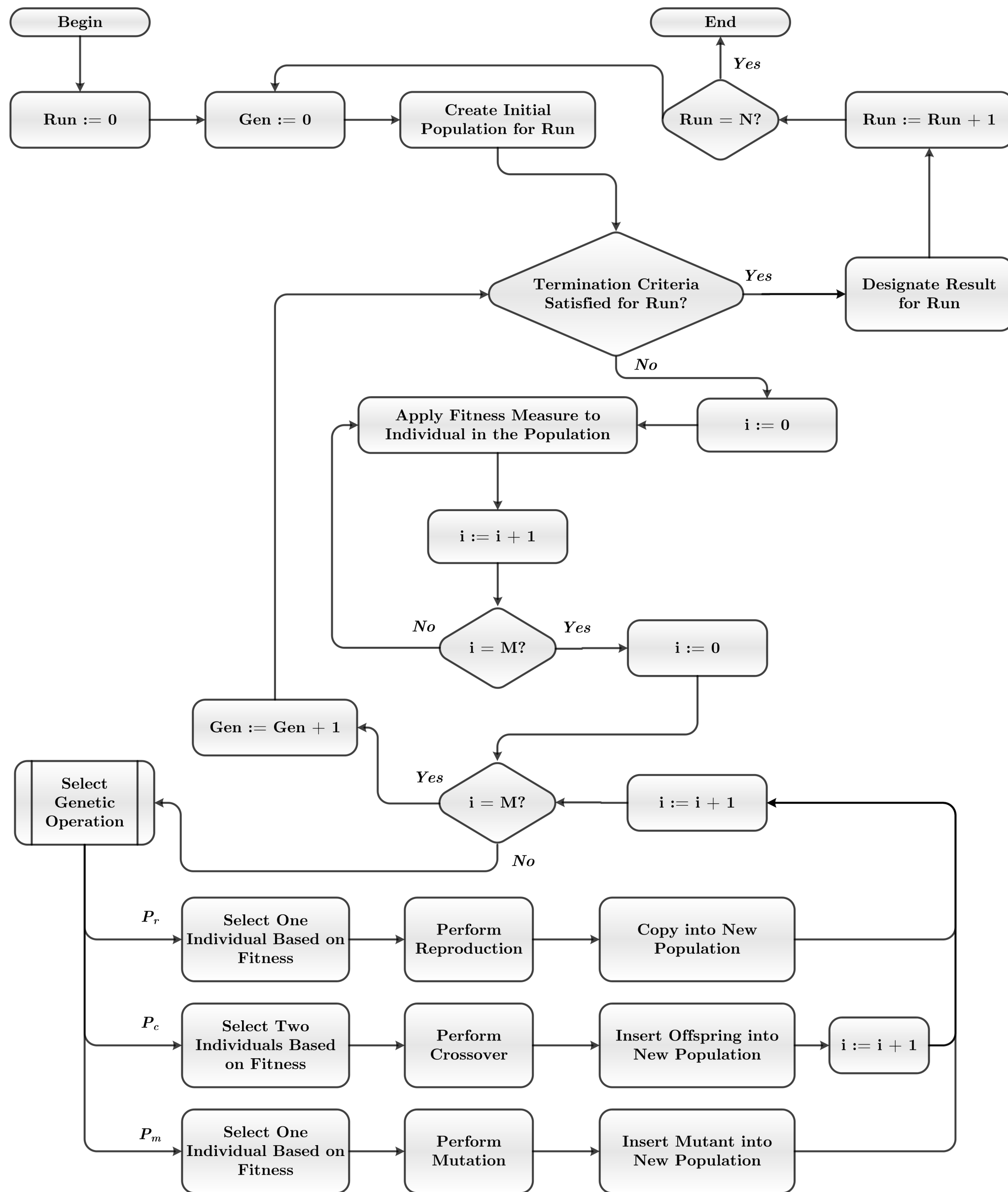

Figure 2.5: Flowchart for the Genetic Programming paradigm [Koz92]. The index i refers to an individual in the population of size M. The variable *Gen* is the number of the current generation. The variable *Run* is the number of the current run, and N is the predefined number of runs.

simplest and earliest are the *Full* and *Grow* methods, and a widely used combination of the two known as *Ramped Half-and-Half* [PLM08]. In both the Full and Grow methods, the initial individuals are generated so that they do not exceed a user specified maximum depth: the Grow method builds random trees depth-first up to a certain depth; the Full algorithm forces full trees up to the maximum depth. With Ramped Half-And-Half, half the initial population is constructed using Full and half using Grow.

GP departs significantly from other evolutionary algorithms in the implementation of the operators for Crossover and Mutation [PLM08]. The most commonly used form of crossover is *Subtree Crossover* (Figure 2.6): given two parents, Subtree Crossover randomly selects a crossover point (i.e., a node) in each parent tree; then, it creates the offspring by replacing the subtree rooted at the crossover point in a copy of the first parent with a copy of the subtree rooted at the crossover point in the second parent. The most commonly used form of Mutation in GP is *Subtree Mutation* (Figure 2.7), which randomly selects a mutation point in a tree and substitutes the subtree rooted there with a randomly generated subtree. Another common form of mutation is *Point Mutation*: a random node is selected and the primitive stored there is replaced with a different random primitive of the same arity taken from the primitive set; if no other primitives with that arity exist, nothing happens to that node. The reproduction operator (as with GAs) simply involves the selection of an individual based on fitness, and the insertion of a copy of this individual in the next generation.

When optimizing computer programs, the most natural way to evaluate their fitness is to execute them and assess their behaviour [Luk09b]. Interpreting a program tree typically means executing the nodes in the tree in an order that guarantees that nodes are not executed before the value of their arguments (if any) is known. This is usually performed by traversing the tree recursively starting from the root node, and postponing the evaluation of each node until the values of its children (arguments) are known [PLM08]; this process is illustrated in Figure 2.8.

The specification of the control parameters in a run is a mandatory preparatory step. There are several parameters; some of the most important include the population size, the probabilities of performing the genetic operations, the minimum and maximum tree sizes, and the stopping criteria. It is impossible to define general guidelines for setting optimal parameter values, as these depend greatly on the details of the application. However, GP is in practice robust, and it is likely that many different parameter values will work [PLM08].

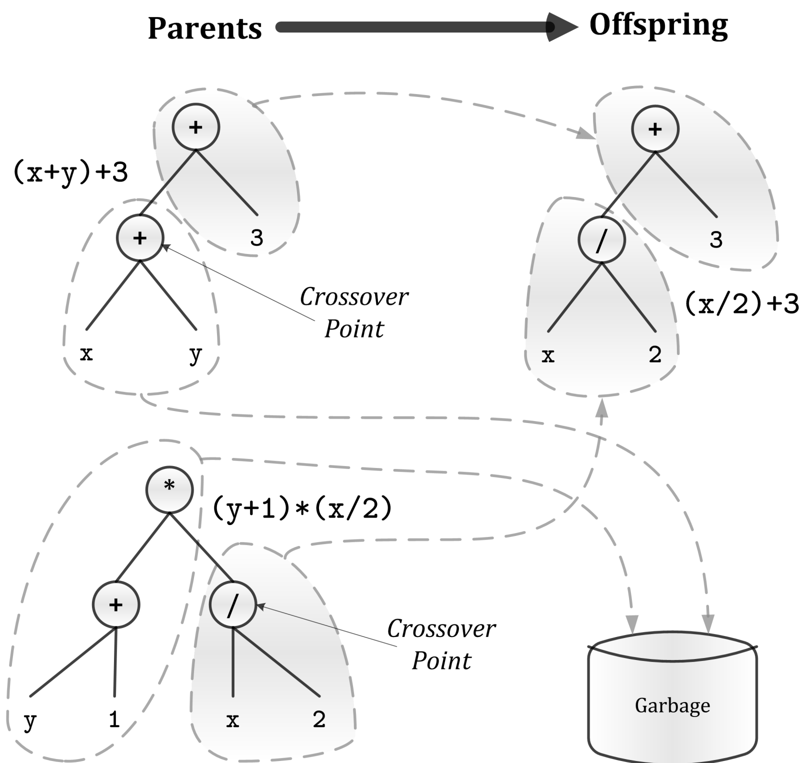

Figure 2.6: Example of GP subtree crossover. Adapted from [PLM08].

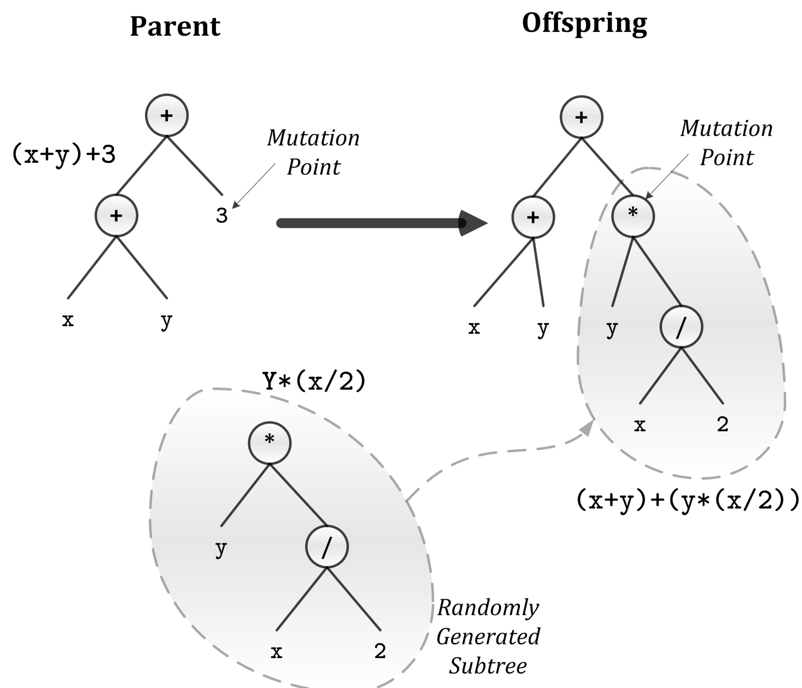

Figure 2.7: Example of GP subtree Mutation. Adapted from [PLM08].

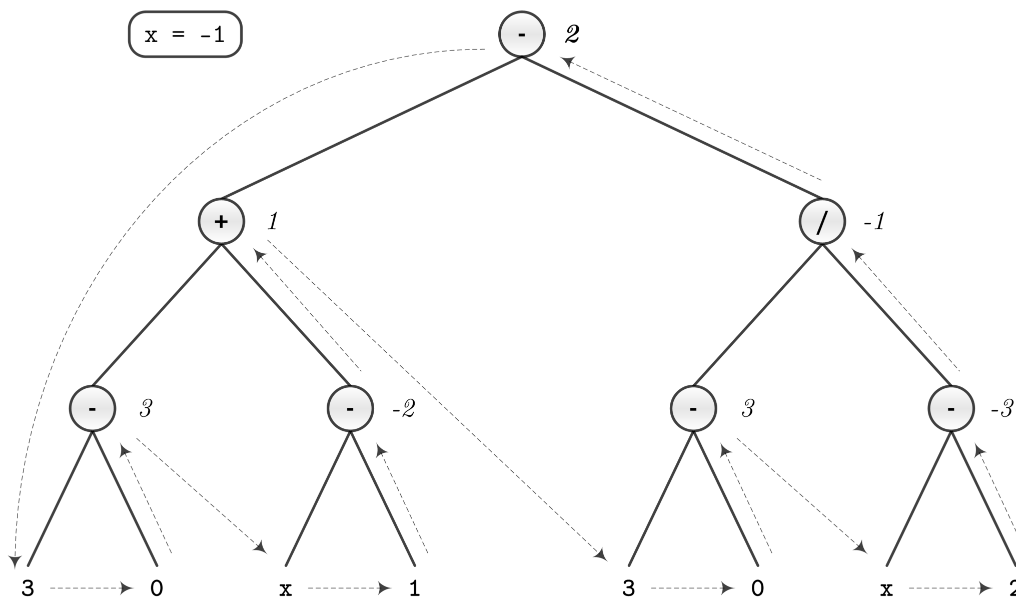

Figure 2.8: Example interpretation of a GP syntax tree (the terminal x is a variable and has a value of -1). The number to the right of each internal node represents the result of evaluating the subtree rooted at that node. Adapted from [PLM08].

Strongly-Typed Genetic Programming

In order for GP to work effectively, most Function Sets are required to have an important property known as "closure" [Koz94], which enables each tree member to be able to process all possible argument values, and ensures that operators will always produce legal offspring.

The closure principle can be broken down into the properties of Type Consistency and Evaluation Safety [PLM08]:

- *Evaluation Safety* is required because many commonly used functions can fail at runtime; an evolved expression might, for example, divide by zero. Evaluation safety is typically dealt with either by: testing functions prior to their execution, and returning a default value if a problem is found; or trapping runtime exceptions and strongly reducing the fitness of programs that generate errors.

- *Type Consistency* is the requirement that all functions have return values of the same type, and that each of their arguments also have this type; it is required to ensure that the output of any subtree can be used as one of the inputs to any node. Universal type compatibility en-

sures that Crossover cannot lead to incompatible connections between nodes, and prevents Mutation from producing illegal programs.

An implicit assumption underlying Type Consistency is that all combinations of structures are equally likely to be useful; in many cases, however, it is known in advance that there are constraints on the structure of the potential solutions. What's more, the nonexistence of types may lead to the generation of syntactically incorrect parse trees; specifically, non-typed GP approaches are unsuitable for representing Object-Oriented programs [HSW96], because any element can be a child node in a parse tree for any other element without having conflicting data types.

There are various distinct approaches to constraining the syntax of the evolved expression trees in GP. The most common are simple structure enforcement, grammar-based constraints, and Strongly-Typed Genetic Programming (STGP). STGP is arguably the most natural approach to incorporate types and their constraints into a GP system [PLM08], since constraints are often expressed using a type system.

In STGP [Mon93], variables, constants, arguments and returned values can be of any data type, with the provision that the data type for each such value is specified beforehand in the Function Set. This allows the initialization process and the genetic operators to only generate syntactically correct parse trees; 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.

The STGP search space is the set of all legal parse trees – i.e., all of the functions have the correct number of parameters of the correct type – and is thus particularly suited for representing the Object-Oriented programs, as it enables the reduction of the search space to the set of *compilable* (i.e., formally feasible [Wap07]) programs, by allowing the definition of constraints that eliminate invalid combinations of operations. What's more, STGP has already been extended to support more complex type systems, including simple generics [Mon95], inheritance [HSW96], and polymorphism [Ols94, Yu01a].

2.4 Evolutionary Testing

Search-Based Software Engineering (SBSE) seeks to reformulate Software Engineering problems as search-based optimisation problems. It has been

applied to a wide variety of Software Engineering areas, including requirements engineering, project planning and cost estimation, automated maintenance, service-oriented software engineering, compiler optimization and quality assessment [Har07b]. Most of the overall literature (an estimated 59% [HMZ09]) in the SBSE area is, however, concerned with Software Testing related applications.

The application of Evolutionary Algorithms to Test Data generation or selection is often referred to as *Evolutionary Testing* [Ton04, WW06b] or *Search-Based Test Data Generation* [McM04]. Evolutionary Testing consists of exploring the space of Test Programs by using metaheuristic techniques that direct the search towards the potentially most promising areas of the input space [Ber07]; its foremost objective is usually that of searching for a set of Test Programs that satisfies a predefined test criterion.

An adequate Evolutionary Testing strategy must generate and select, in a systematic manner and at a reasonable computational cost, only those Test Programs that are relevant and are expected to be fault revealing. The search objective must 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 [Har07b].

The *effectiveness* of a Test Data generation technique is closely related to the quality of the resulting Test Set; the generated Test Programs should ideally achieve full structural coverage of the Test Object and/or meet the specified functional requirements, depending on the testing strategy selected. Conversely, a technique's *efficiency* is related to the speed of the technique to converge towards the test objective; it is generally measured either to assess whether it can be used in practice or to compare two methodologies. Some common efficiency measures used in the Evolutionary Testing domain include the number of iterations or fitness evaluations required to find the best solution, the time spent performing the search, and the size of the resulting Test Set [ABHPW08].

Evolutionary Algorithms have already been applied with significant success to the search for Test Data; Xanthakis *el al.* [XES+92] presented the first application of heuristic optimization techniques to Test Data generation in 1992. However, research has been mainly geared towards generating Test Data for procedural software, and traditional methods – despite their effectiveness and efficiency – cannot be applied without adaptation to Object-Oriented systems.

The application of search-based strategies to Unit Testing of Object-

Oriented programs is, in fact, fairly recent – the first approach was presented in 2004 by Tonella [Ton04] – and is yet to be investigated comprehensively [HHL+07]. This is mostly because Object-Oriented Evolutionary Testing is particularly challenging: in an Object-Oriented system, the basic test unit is a class instead of a subprogram, and testing should hence focus on classes and objects; and while a Test Program for procedural software typically consists of a sequence of input values to be passed to a procedure upon execution, Test Programs for class methods must also account for the state of the objects involved in the methods' calls. It is, in fact, a difficult subject, especially if the aim is to implement an automated solution, viable with a reasonable amount of computational effort, which is adaptable to a wide range of Test Objects; even though several approaches have been studied to address this problem, a system that is able to generate an optimum set of Unit Tests for any generic Test Object is yet to be developed [AY08b].

2.4.1 Object-Oriented Evolutionary Testing

Software Testing can benefit from Object-Oriented technology – for instance, by capitalising 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 myth that the enhanced modularity and reuse brought forward by the Object-Oriented programming paradigm could prevent the need for testing has long been rejected [Ber07]. In fact, the Object-Oriented paradigm poses several hindrances to testing due to some aspects of its very nature [BS94]:

- *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* – polymorphism and dynamic binding call for new coverage models, and 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.

The hidden state, in particular, poses a serious barrier to the Object-Oriented Software Testing. This issue – usually referred to as the *State Problem* [MH03] – is related with the fact that, due to the encapsulation principle of the Object-Oriented paradigm, the state of an object is accessible only through an interface of public methods. As such:

- the only way to change the state of an object is through the execution of a series of method calls (i.e., it is not possible to directly manipulate the object's attributes);

- and the only way to observe the state of an object is through its operations, which hinders the task of accurately measuring the quality of a candidate Test Program.

The term *Object-Oriented Evolutionary Testing* usually refers to the search-based Unit Test generation for Object-Oriented Software [HMZ09], and involves the search for Unit Test programs that define interesting state scenarios for the objects involved in the call to the *Method Under Test (MUT)*. During Test Program execution, all participating objects must be created and put into particular states by calling several instance methods on these objects. The search space thus encompasses the set of all possible inputs – and their states – to the public methods of a particular *Class Under Test (CUT)*, including the implicit parameter (i.e., the `this` parameter) and all the explicit parameters.

A Test Program for Object-Oriented software typically consists of a *Method Call Sequence (MCS)*, which represents the test scenario. In general, a MCS is a sequence of method calls, constructor calls and value attributions, when assuming that no decision or repetition structures are present [Wap07]. Given that each MCS usually focuses on the execution of one particular method (the MUT), at least one method call must refer to that method – in general, the last element of the sequence. Also, as was made clear by the previous examples, it is usually not possible to test a single class in isolation; other data types may be necessary for calling the CUT's public methods. The set of classes which are relevant for testing a particular class is called the *Test Cluster*. In summary, a Unit Test for Object-Oriented programs usually requires [WL05]:

- 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.

Let us consider the `search` method of the `Stack` class of Java Development Kit (JDK) 1.4 (Listing 2.1 on page 17) as an example. The `Stack` container class represents a last-in-first-out stack of objects, which extends the class `Vector`** with five operations (`push`, `pop`, `empty`, `peek` and `search`) that allow a vector to be treated as a stack; the `search` method returns the 1-based position (i.e., the distance from the top) where an object is on the stack, with the `equals` method being used to compare the `Object`†† instance provided as a parameter to the items in the stack.

The behaviour of the `search` method differs depending on both the state of the stack on which the method call is issued (i.e., empty or containing elements) and on the properties of the `Object` instance passed to the method as an argument (i.e., the stack may either contain it or not). If White-Box criteria are considered, and in order to achieve full structural coverage of the `search` MUT (i.e., in order to traverse all the nodes, edges or branches of a CFG representing the method, e.g., the one depicted in Figure 2.4 on page 17), a Unit Test generation framework must be able to generate a Test Set encompassing all of the aforementioned state scenarios.

Modifying and "tuning" the state of the `Stack` and `Object` instances, however, is not trivial. The state of the stack can only be modified by calling one of the 5 public methods made available by its public Application Programming Interface (API), and these methods have method call dependencies themselves (e.g., an `Object` instance must be created and passed to the `push` method in order to issue a method call).

What's more, some of these methods are not state-changing (namely `empty`, `peek`, and `search` itself), an could therefore be safely discarded because they provide no aid to the search; a systematic approach to perform this task will be described in Chapter 6. Also, the search for `Object` instances in a stack is problematic because `Object`'s `equals` method implements the most discriminating possible equivalence relation: two `Object`

**http://java.sun.com/j2se/1.4.2/docs/api/java/util/Vector.html
††http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html

references are only considered equal if they both refer to exact same object. Therefore, in order to successfully find an `Object` instance in a `Stack`, the Test Data generation framework should be able to reuse object instances; a methodology for implementing this feature on GP-based Evolutionary Testing frameworks is detailed in Chapter 8.

Listing 2.4 depicts an example Test Program for Object-Oriented software; the MUT is the `search` method of the `Stack` class. In this program, instructions 1, 3 and 5 instantiate new objects, whereas instructions 2 and 4 aim to change the state of the `stack0` instance variable that will be used, as the implicit parameter, in the call to the MUT at instruction 6.

It should be noted that syntactically correct and compilable Test Programs may still abort prematurely, if a runtime exception is thrown during execution [WW06a]. In the example Test Program shown in Listing 2.4, instruction 2 will throw a runtime exception (an `EmptyStackException`[‡‡], to be precise), rendering the Test Program unfeasible; when this happens, it is not possible to assess the quality of the Test Case because the final instruction (i.e., the call to the MUT) is not reached. Test Programs can thus be separated in two classes:

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

Unfeasible Test Programs should be penalised when searching for an adequate Test Set; however, if unfeasible Test Cases are blindly discarded, the definition of complex Test Programs will be discouraged because, in general, longer and more intricate Test Programs are more prone to throw runtime exceptions. A Test Program evaluation strategy which handles this issue will be presented in Chapter 5; also, in Chapter 7, an adaptive strategy for favouring the selection of instructions that do not throw runtime exceptions (among other metrics) will be explored.

Test Data generation by means of Evolutionary Algorithms requires the definition of a suitable representation of Object-Oriented Test Programs; however, it is still a very young field of study, and no conclusions have been reached on deciding what is the best search algorithm that should be applied to it. Neverteless, nature-inspired algorithms seem to perform better

[‡‡] http://java.sun.com/j2se/1.4.2/docs/api/java/util/EmptyStackException.html

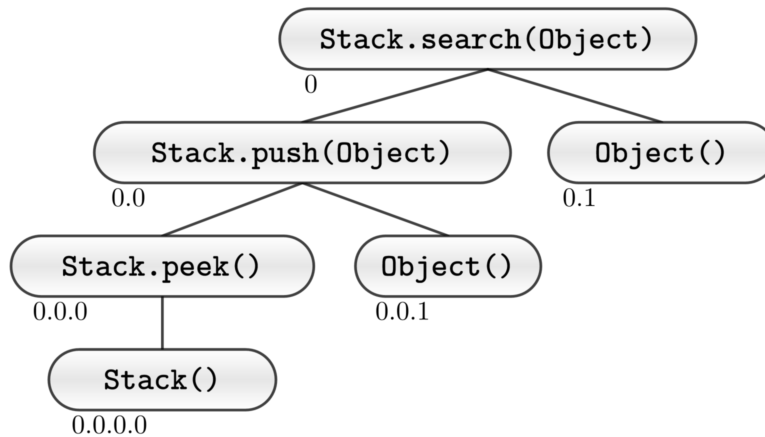

Figure 2.9: Example Method Call Tree. The Method Under Test is the `search` method of the `Stack` class.

```
0.0.0.0 Stack()
0.0.0 Stack.peek() [0.0.0.0 Stack]
0.0.1 Object()
0.0 Stack.push(Object) [0.0.0 Stack.peek(), 0.0.1 Object()]
0.1 Object()
0 Stack.search(Object) [0.0 Stack.push(Object), 0.1 Object()]
```

Listing 2.3: Example Method Call Sequence, resulting from the linearisation of the Method Call Tree depicted in Figure 2.9.

```
Stack stack1 = new Stack();
stack1.peek();
Object object2 = new Object();
stack1.push(object2);
Object object3 = new Object();
stack1.search(object3);
```

Listing 2.4: Example Test Program, synthesised with basis on the Method Call Sequence depicted in Listing 2.3.

than "traditional" techniques (e.g., based on symbolic execution and state matching) because they seem able to solve more complex test problems in less time [AY08b]. Existing evolutionary approaches to Test Data generation have been mainly focused on the usage of GAs and GP (cf. Chapter 3), even though experiments have also been performed with basis on another metaheuristics (e.g., Ant Colony Optimization, Hill Climbing, Simulated Annealing and Memetic Algorithms). Among these, GP emerges as a natural candidate to address Object-Oriented Evolutionary Testing problems for various reasons, which include:

- GP is usually associated with the evolution of tree structures (conversely, GAs typically evolve binary or real number strings). It is thus particularly suited for representing and evolving Test Programs, which may be represented as Method Call Trees (cf. Chapter 4).

- There are a number of typing mechanisms available for GP – most notably STGP [Mon95] – which facilitate the encoding of Object-Oriented programs.

- A GP tree can vary in length throughout the run, thus allowing experimenting with different sized Test Programs; on the other hand, in traditional GAs, the length of the binary string is typically fixed before the solution procedure begins [SD07].

- Because GP evolves active structures the solutions can be executed without post-processing, while GAs typically operate passive structures (e.g, binary strings) which require post-processing [SD07].

- In STGP, the Function Set defines the constraints involved in the construction of the solutions – i.e., it contains the set of instructions from which the algorithm can choose when building Test Programs – and is possible to systematically define the Function Set solely with basis on Test Cluster information, and to automate the analysis and parameterisation processes (cf. Chapter 4).

The following Chapter will be devoted to presenting related work in the area of Search-Based Test Data Generation, with the emphasis being put on Object-Oriented Evolutionary Testing; next, our own GP-based approach to subject will be presented; and in subsequent Chapters, several methodologies for enhancing the performance of Evolutionary Testing methodologies will be proposed and discussed.

2.5 Summary

Software Testing is the process of exercising an application to detect errors and to verify that it satisfies the specified requirements. When performing Unit Testing, the goal is to warrant the robustness of the smallest units – the Test Objects – by testing them in an isolated environment. Unit Testing is performed by executing the Test Objects in different scenarios using relevant and interesting Test Programs (or 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. A Unit Test Case for Object-Oriented software consists of a MCS, which defines the test scenario. During Test Program execution, all participating objects are created and put into particular states through a series of method calls. Each Test Case focuses on the execution of one particular public method – the MUT.

Most work in testing has been done with "procedure-oriented" software in mind; nevertheless, traditional methods cannot be applied without adaptation to Object-Oriented systems. For Object-Oriented programs, classes and objects are typically considered to be the smallest units that can be tested in isolation. An object stores its state in fields and exposes its behaviour through methods. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation – a fundamental principle of Object-Oriented programming.

Evolutionary Algorithms use simulated evolution as a search strategy to evolve candidate solutions for a given problem, using operators inspired by genetics and natural selection. GP, in particular, is a specialization of GAs usually associated with the evolution of tree structures; it focuses on automatically creating computer programs by means of evolution, and is thus especially suited for representing and evolving Test Programs. The nodes of a GP tree are usually not typed – i.e., all the functions are able to accept every conceivable argument. Non-typed GP approaches are, however, unsuitable for representing Test Programs for Object-Oriented software; conversely, STGP allows the definition of types for the variables, constants, arguments and returned values. The only restriction is that the data type for each element must be specified beforehand in the Function Set. This causes the initialization process and the various genetic operations to only construct syntactically correct trees.

The application of Evolutionary Algorithms to Test Data generation is often referred to in the literature as Evolutionary Testing. The goal of Evolutionary Testing problems is to find a set of Test Cases that satisfies a certain

test criterion – such as full structural coverage of the Test Object. The test objective must be defined numerically and suitable fitness functions, that provide guidance to the search by telling how good each candidate solution is, must be defined. The search space is the set of possible inputs to the Test Object; in the particular case of Object-Oriented programs, the input domain encompasses the parameters of the Test Object's public methods. As such, the goal of the evolutionary search is to find Test Programs that define interesting state scenarios for the variables which will be passed, as arguments, in the call to the MUT. One of the most pressing challenges faced by researchers in the Evolutionary Testing area is the State Problem, which occurs with objects that exhibit state-like qualities by storing information in fields that are protected from external manipulation – and that can only be accessed through the public methods that expose the classes' internals and grant the access to the objects' state.

Defining a Test Set that achieves full structural coverage may, in fact, involve the generation of complex and intricate Test Cases in order to define elaborate state scenarios, and requires the definition of carefully fine-tuned methodologies that promote the traversal of problematic structures and difficult control-flow paths.

Chapter 3

Related Work

Metaheuristics have already been applied with significant success to the generation of Test Data; in this Chapter, related work on Evolutionary Testing is explored. The focus is put on evolutionary approaches for the structural Unit Testing of Object-Oriented programs. Firstly, GA-based techniques are described. A discussion on methodologies which employ the GP technique follows. Finally, special attention is paid to approaches with employ other metaheuristic strategies.

3.1 Object-Oriented Evolutionary Testing Techniques

Miller and Spooner are typically considered the first to combine the results of actual executions of a program with a search technique in 1976; in [MS76], numerical maximisation is utilised as a technique for generating floating-point Test Data. Xanthakis *et al.* [XES+92], in 1992, were the first to apply Evolutionary Algorithms to generate structural Test Data; GAs were employed to generate Test Data for structures not covered by Random Search.

However, research has been mainly geared towards generating Test Data for Procedural Software. The first approach to the field of Object-Oriented Evolutionary Testing was presented by Tonella in 2004 [Ton04].

3.1.1 Genetic Algorithms-based Approaches

In [Ton04], a technique for automatically generating input sequences for the structural Unit Testing of Java classes by means of GAs was proposed. Possible solutions are represented as chromosomes, which consist of the input values to use in Test Program execution; the creation of objects is also accounted for (Listing 3.1). Because the GA performs on chromosomes with a specific organization, the standard Crossover and Mutation operators cannot be applied; special Mutation operators (for replacing input values, changing constructors, and inserting/removing method invocations) and a one-point Crossover operator are defined. A population of Test Cases is evolved in order to increase a measure of fitness accounting for their ability to satisfy a branch coverage criterion; new Test Programs are generated as long as there are targets to be covered or a maximum execution time is reached. The *eToc* framework for the Evolutionary Testing of Object-Oriented software was implemented and made available* as a result of this research. Experimental studies were performed on the StringTokenizer, BitSet, HashMap, LinkedList, Stack and TreeSet JDK 1.4 classes; full branch coverage was not achieved in all of them, but the only branches remaining corresponded to non-traversable portions of code. Even though several Evolutionary Testing-related problems were not addressed on this work (e.g., the usage of universal Evolutionary Algorithms, encapsulation, complex state problems, Test Program feasibility, search guidance, MCS minimisation), it was able to prove the applicability of Evolutionary Algorithms to Test Data generation.

Tonella's research inspired Object-Oriented Evolutionary Testing literature; several approaches built on his experiments with GAs in the following years.

In [WL05] the focus was put on defining a grammar-based encoding for Test Programs which enabled the application of any given universal Evolutionary Algorithms (e.g., Hill Climbing [RNC+96] or Simulated Annealing [LA87]) to Object-Oriented Evolutionary Testing; unlike Tonella's previous approach, this methodology allows an effortless change of the evolutionary strategy employed. Objective functions based on the distance-oriented approach, which guide the evolutionary search in cases of conditions which are hard to meet by random, are also defined. However, the technique proposed permits the generation of individuals which cannot be decoded into Test Programs without errors; this hindrance is circumvented by the

*http://star.itc.it/etoc/

```
1  <chromosome> ::= <actions> @ <values>
2  <actions>    ::= <action> {: <actions>}?
3  <action>     ::= $id = constructor ( {<parameters>}? )
4                | $id = class # null
5                | $id . method ( {<parameters>}? )
6  <parameters> ::= <parameter> {, <parameters>}?
7  <parameter>  ::= builtin-type {<generator>}?
8                | $id
9  <generator>  ::= [ low ; up ]
10               | [ genClass ]
11 <values>     ::= <value> {, <values>}?
12 <value>      ::= integer
13               | real
14               | boolean
15               | string
```

Listing 3.1: Syntax of chromossomes utilised by eToc [Ton04].

definition of a fitness function which penalises invalid sequences. Experiments were performed on a custom-made Java class (StateCounter); even though coverage metrics were not provided, relevant results included the observation that the number of inconvertible individuals visibly decreased constantly over the generations.

Cheon *et al.* [CKP05] proposed combining the Java Modelling Language (JML) [LBR06] and GAs in order to automate Test Data generation for Java programs. JML is used both as a tool for describing test oracles and as a basis for generating Test Data; each class to be tested is assumed to be annotated with JML assertions. A proof-of-concept tool is briefly described with basis on a custom-made example. In [CK06a], a specification-based fitness function for evaluating boolean methods of Object-Oriented programs was presented, with an example being provided for illustration and experimentation purposes. The evolutionary search's efficiency was reported to improve from 300% up to 800% as a result of application of the fitness function.

Inkumsah and Xie [IX07] introduced a technique which merges Concolic Testing (a combination of concrete and symbolic testing techniques [Sen07]) and Evolutionary Testing; this approach was implemented into the Evacon framework, which integrated Tonella's eToc Evolutionary Testing tool and the jCUTE [SA06] Concolic Testing tool (which tests Java classes using the dynamic symbolic execution technique). Evolutionary Testing is used to search for desirable method sequences, while Concolic Testing is employed

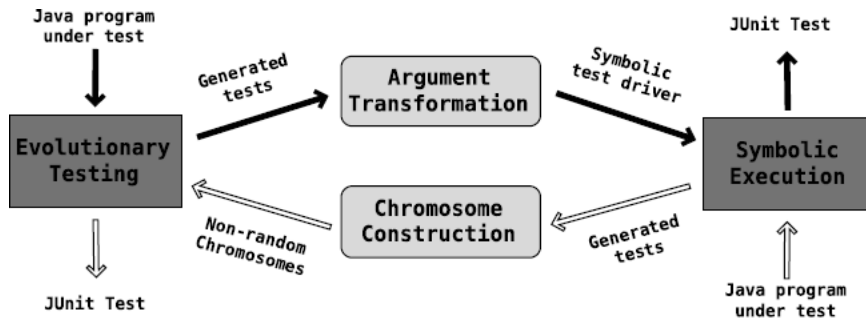

Figure 3.1: Evacon framework overview [IX08].

to generate desirable method arguments. The inclusion of Concolic Testing into the process was supported by the perception that typical Evolutionary Testing tools do not use program structure or semantic knowledge to directly guide test generation, nor provide effective support for generating desirable primitive method arguments. Empirical studies were conducted on 6 Java classes taken from the standard library, with the results showing that the tests generated using Evacon achieved higher branch coverage than Evolutionary Testing or Concolic Testing alone.

The Evacon tool is described with further detail in [IX08]. Additionally, Evacon is empirically compared to eToc, jCUTE [SA06], JUnit Factory[†] (an industrial test generation tool developed by AgitarLabs), and Randoop [PE07] (a Random Testing tool). Evacon is reported to achieve higher branch coverage than any of the aforementioned tools for the 13 Java classes tested. Figure 3.1 shows an overview of Evacon, which includes 4 components: Evolutionary Testing, symbolic execution, argument transformation (for bridging from Evolutionary Testing to symbolic execution), and chromosome construction (for bridging from symbolic execution to Evolutionary Testing). In a short position paper [XTdHS08], Evacon's authors briefly describe an additional tool for the generation of method sequences with a demand-driven mechanism and a heuristic-guided mechanism, which is incorporated into Pex [Til08] (a White-Box Test Data generation framework for .NET[‡]).

In [DJAR07], Dharsana *et. al* briefly describe a GA-based tool for generating Test Cases for Java Programs. Experiments were performed on binary tree, linkedlist, bubble sort (the source of the classes is not provided), and two custom-made programs, but no details were provided on the setup or

[†] http://www.junitfactory.com
[‡] http://research.microsoft.com/en-us/projects/pex/

results.

The work of Ferrer *et. al* [FCA09] proposes dealing with the inheritance feature of Object-Oriented programs by focusing on the Java `instanceof` operator. The main motivation is that of providing guidance for an automatic Test Case generator in the presence of conditions containing the aforementioned operator, and is supported by the fact that it appears in 2700 of the 13000 classes of the JDK 1.6 class hierarchy. Two Mutation operators, that change the solutions based on a distance measure which computes the branch distance in the presence of the `instanceof` operator, were proposed, and included in the GA-based "Evolutionary Solver" described. Experiments were performed on nine custom-made Test Programs, each consisting of one method with six conditions; the Mutation operators proposed were reported to behave well when used in place of a simpler mutation operator, and when compared to Random Search. Future plans involve combining the approach proposed with other Evolutionary Testing approaches, and experimenting with real-world software.

3.1.2 Genetic Programming-based Approaches

The first GP-based approaches to Object-Oriented Evolutionary Testing were presented in 2006 by Wappler and Wegener [WW06b, WW06a], and Seesing and Gross [SG06, See06].

The encoding of potential solutions using the STGP technique was first proposed in [WW06b]. Test Programs are represented as STGP trees, which are able to express the call dependencies of the methods that are relevant for a given Test Object. In contrast with previous approaches in this area, neither repair of individuals nor penalty mechanisms are required in order to achieve sequence validity; the usage of STGP preserves validity throughout the entire search process (i.e, only compilable Test Programs are generated by tree builders and genetic operators). 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 Function Set is derived from the signatures of the methods of the Test Cluster classes, and the Type Set is derived from the inheritance relations of the Test Cluster classes. Runtime exceptions are dealt with by means of a distance-based fitness function. Experiments were performed on four JDK 1.3 classes (Stack, BitSet, BoolStack, ObjectVector), with full structural coverage being achieved in all cases.

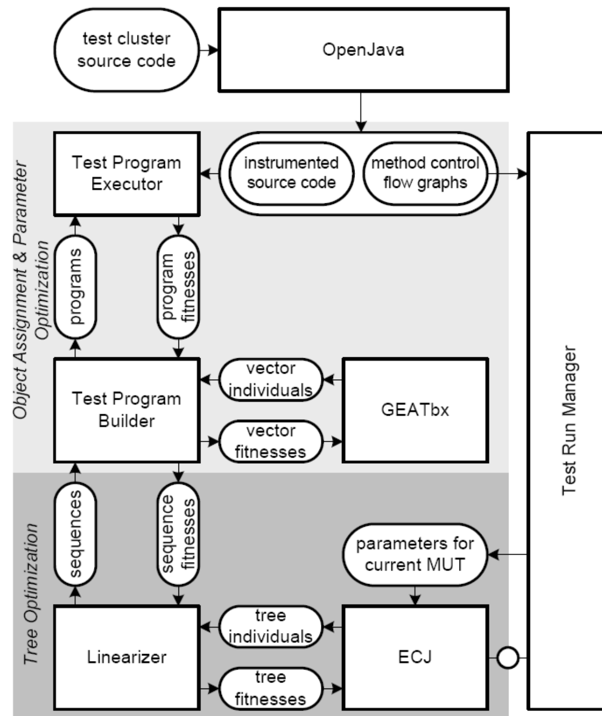

Figure 3.2: EvoUnit framework overview [WW06a].

In [WW06a], Wappler and Wegener extended their previous work and focused on dealing with unfeasible Test Programs; unlike previous approaches, the search is guided in case of uncaught runtime exceptions. They propose a minimising distance-based fitness function in order to assess and differentiate the Test Programs generated during the evolutionary search, which rates them according to their distance to the given test goal (i.e., the program element to be covered); the aim of each individual search is therefore to generate a Test Program that covers a particular branch of the CUT. This fitness function makes use of a distance metric that is based on the number of non-executed methods of a Test Program if a runtime exception occurs. The EvoUnit framework (Figure 3.2), which implements the concepts proposed in [WW06b, WW06a], is also described; unfortunately the tool is proprietary and is thus not openly available. Experiments were performed on a custom-made Test Cluster (Controller and Config), with full branch coverage being achieved.

Wappler *et. al* suggest an improvement of their Evolutionary Testing approach in [WS07], which particularly addresses the test of non-public methods. The existing objective functions are extended by an additional component that accounts for encapsulation; candidate Test Programs are

rewarded if they cover calls to specific non-public methods. Experiments performed of 6 Java classes taken from the JDK 1.4, the Quilt project 0.6a5[§], and JFreeChart 1.0.1[¶] yield better branch coverage for non-public methods in comparison with Random Search and their previous approach.

In his Ph.D. Thesis [Wap07], Wappler provides a thorough explanation of his approach to automatic test generation for Object-Oriented software, and compares it to other testing techniques (e.g., symbolic execution [Kin76] and constraint solving [Tsa93]). An empirical investigation also demonstrated the effectiveness of the methodology; it outperformed random testing and two commercial test sequence generators (CodePro[‖] and Jtest[**]) when being allocated the same resources. Limitations on the current stage of development of the approach were also pinpointed: the efficiency level of the approach decreases as the Test Cluster (and, in consequence, the Function Set) increases in size; and the test sequences might include unnecessary method calls.

Our approach to Object-Oriented Evolutionary Testing is also STGP-based and, as such, builds on the work of Wappler *et. al.* We have focused our studies on automating both the Test Object analysis [RdVZR07, RZdV07a] and the Test Data generation [RZdV07a, Rib08] processes, and on presenting novel contributions for search guidance [RZdV07b, RZRdV08a, RZRFdV09], Input Domain Reduction [RZRdV08b, RZRFdV09], Adaptive Evolutionary Testing [RZRdV10a] and Object Reuse [RZRdV10b]. These topics (and relevant related work) will be discussed in the following Chapters.

Seesing and Gross proposed a distinct typed GP mechanism for creating Test Data for Object-Oriented systems; in [SG06], the advantages of employing a tree-shaped data structure (which can be mapped instantly to the abstract syntax trees commonly used in computer languages) for representing Test Programs is discussed, and the proposed GP methodology is compared to previous GA-based approaches (namely, [Ton04, WL05]). A custom-made encoding of Object-Oriented Test Programs is presented, and Mutation operators for method introduction, method removal, and variable introduction are described. Experiments were performed on 5 Java Classes: BitSet, HashMap, TreeMap, XMLElement and StringTokenizer. The results demonstrated the advantage of GP over Random Search, with much

[§]http://quilt.sourceforge.net/

[¶]http://www.jfree.org/jfreechart/

[‖]http://www.instantiations.com/codepro/

[**]http://www.parasoft.com/jsp/products/jtest.jsp

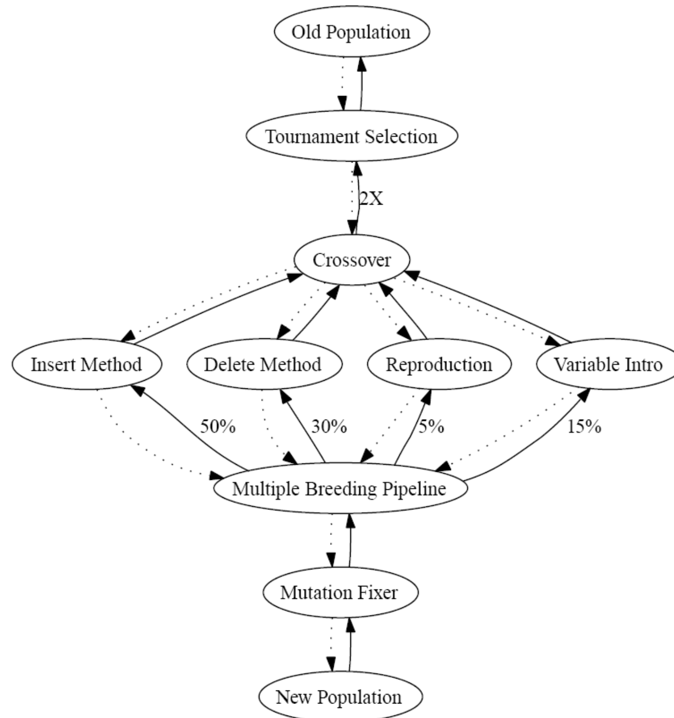

Figure 3.3: Breeding pipeline used by EvoTest [See06].

higher structural coverage being achieved. In his Master Thesis [See06], Seesing elaborates on the approach and describes the EvoTest Test Case generation and Sofware analysis framework (Figure 3.3).

A GP-based Object-Oriented Evolutionary Testing methodology was also presented in [GR08], which is in the line of the work developed by Seesing *et. al.* The encoding and decoding of Test Program into changeable data structures are discussed, and experiments are performed on 5 classes of the open source Java project HTMLParser[††]. The approach is reported to behave well when compared to Random Testing.

Arcuri and Yao employed STGP in a different scenario. [AY07a] introduces the idea of employing co-evolution [Hil90] for automatically generating Object-Oriented programs from their specification; STGP is used to evolve these programs and, at the same time, the specifications are exploited in order to co-evolve a set of Unit Tests. More specifically, given a specification of a program, the goal is to evolve a program that satisfies it; at each step of the evolutionary process, each program is evaluated against a set of Unit Tests that also depends on the specification. The more Unit Tests a

[††]http://htmlparser.sourceforge.net/

program is able to pass, the higher its fitness will be; similarly, Unit Tests are rewarded on how many programs they make fail. The experiments performed on 4 array-related problems (sorting, searching for the highest value, searching for the highest occurrence and testing whether all the elements are identical) achieved successful results. In [Arc08, AY08a] the authors elaborate on the topic and provide further details on the approach and corresponding tool, and on [AWCY08] they present a related co-evolution approach to optimising software, which also involves Multi-Objective Optimisation [YH07]; still, and even though it is argued that it possible to apply the methodology proposed to any problem that can be defined with a formal specification, its application to the Object-Oriented software was not the subject of the latter study.

Even thought the GP technique is clearly applicable to the Evolutionary Testing of Procedural Software, it has not been explored in the published formulations of the problem [HMZ09]. Other studies within the scope of SBSE have, however, investigated its use: Evett *et al.* [EKdCA98] employed GP for software quality prediction; Burges and Lefley [BL05] applied GP to the estimation of a software project effort; and Katz and Peled [KP08a, KP08b] provided a Model Checking-based GP approach for verification and synthesis from specification.

3.1.3 Other Metaheuristics-based Approaches

Even thought the majority of the Object-Oriented Evolutionary Testing literature is devoted to the study of either GAs or GP, there are several studies that focus on distinct evolutionary techniques. In fact, as stated in [AY07c], other metaheuristic techniques have the potential to achieve promising results in this area.

An approach which employed a hybrid of Ant Colony Optimization [DS04] and Multi-Agent GAs [ZLXJ03] was the subject of [LWL05]. The focus was on the generation of the shortest MCS for a given test goal, under the constraint of state dependent behaviour and without violating encapsulation. This hybrid algorithm (called Ant PathFinder) was reported to yield encouraging results on the experiments performed on the CalendarScheduler class (a data structure included in the discrete event simulator NS2 2.26) and on Red-Black Tree [CLRS01].

Sagarna *et. al* [SAY07] addressed the Object-Oriented Evolutionary Testing problem using Estimation of Distribution Algorithms [LL02]. Estimation of Distribution Algorithms only differ from GAs in the procedure

to generate new individuals; instead of using the typical breeding operators, Estimation of Distribution Algorithms perform this task by sampling a probability distribution previously built from the set of selected individuals. The focus was put on generating Test Data for container classes; experiments were performed on Vector, LinkedList and Hashtable, extracted from JDK 1.4. Relevant conclusions include the observations that the positions at which methods are called in the Test Program are (considering the particular conditions of the approach) independent of each other, and that coverage grows as the length of the MCS increases.

In [LR08] Liaskos *et. al* investigated whether the properties of the Clonal Selection algorithm [dCZ02] (memory, combination of local and global search) could help tackling the hindrances posed by Object-Oriented Evolutionary Testing. Clonal Selection is one of the most popular population-based Artificial Immune Systems algorithms [dCT03] (computational systems inspired by theoretical immunology and observed immune functions). Despite employing Mutation to generate new populations, and unlike GAs, Clonal Selection performs Mutation on the selected solutions with a rate that is inversely proportionate to their fitness, and does not use crossover; also, high quality solutions are stored for future use, leading to a faster immune response. The encoding of solutions is identical to the one used by the GAs (Test Cases are encoded as chromosomes); the goal is to minimise the distance between "receptors" (i.e., the executed paths in the CFG) and "antigens" (i.e., the test targets). Comparative experiments were performed on 6 Java classes (those used in [Ton04]) to assess the behavior of the hybridisation of a GA with both Artificial Immune Systems and Local Search. The results suggested that hybridised approaches usually outperform the GA; however, there are scenarios for which the hybridisation with Local Search is more suited than the more sophisticated Clonal Selection algorithm. This paper extended the authors' previous works ([LRW07, LR07]), which also to addressed the problem of automated testing with data-flow as the adopted coverage criterion.

Most of the research of Arcuri *et. al* in the Object-Oriented Evolutionary Testing area is related with investigating the application of distinct search algorithms to the Test Data generation for container classes (i.e., classes designed to store any arbitrary type of data). This is precisely the topic of [AY07b]. Hill Climbing, GAs and Memetic Algorithms [Mos89] were the evolutionary approaches used and compared (extending their previous work presented in [AY07d]). While GAs are global metaheuristics and Hill Climbing is a local search metaheuristic, Memetic Algorithms can approximately

be described as a population-based metaheuristics in which, whenever an offspring is generated, a local search is applied to it until it reaches a local optimum. Novel branch distances for handling disjunctions and conjunctions of predicates, new search operators, and a way to reduce the search space for Object-Oriented software were also presented. The test criterion was White-Box Testing, with the goal being to achieve branch coverage. Case studies were conducted on the Stack, Vector, LinkedList, Hashtable and TreeMap classes of JDK 1.4, which showed that the Memetic Algorithm outperforms the other algorithms; also, the novel search operators and the search space reduction technique were able to increase its performance.

In [AY08b], the authors elaborate on their previous studies, and focused on the difficulties of testing Object-Oriented container classes with metaheuristic search algorithms. It is stressed out that container classes are particularly important given that, even if a testing tool is designed for handling a generic program, container classes are often used as benchmarks. Relevant contributions include Input Domain Reduction [HHL+07] and Testability Transformation [HHH+04] techniques, and addressing the MCS length minimisation problem. The performance of five search algorithms (Random Search, Hill Climbing, Simulated Annealing, GAs and Memetic Algorithms) was compared on 7 container classes (Vector, Stack, LinkedList, Hashtable and TreeMap from the JDK 1.4; and BinTree and BinomialHeap from JPF [VPP06]). The experimental studies revealed TreeMap (an implementation of Red-Black Tree) as the most difficult Container to test, with Memetic Algorithms arising as the best technique for the problem; also interestingly, Hill Climbing performed better than GAs (local Search algorithms are generally supposed to behave worse in these situations [WBS01]), and Random Search behaved poorly especially on more complicated problems.

In [Arc09b], Arcuri carries out a set of theoretical analyses regarding the usage of metaheuristic search techniques in Software Testing. The focus is on assessing if (and why) a search algorithm is effective on a Software Testing problem. This work compares four distinct search algorithms: Random Search, Hill Climbing, (1+1) Evolutionary Algorithm, and GA. For this purpose, the `put` and `remove` methods of the TreeMap Java class were selected as Test Objects, and a White-Box scenario, in which the full branch coverage was sought, was considered. For the sake of simplicity, only integer objects were used, the same object was used for both as the key and inserted object, and insertion/removal of null objects was not considered. Also, constraints were put on the input ranges and on the number of method calls in the test sequences. The results yielded by this study indicated

that a simple (1+1) Evolutionary Algorithm performs better than a fairly tuned GA – in contradiction with the current trend, in which population algorithms are very common. Also, in this case, even Random Search was able to give optimal solutions in a reasonable time when constraints were used. As future work, the author mentioned the importance of extending the analyses to scenarios in which no constraints are defined.

In this Ph.D. Thesis [Arc09a], Arcuri compiles and elaborates on his previous proposals. Relevant contributions to the SBSE area include theoretical analyses of search algorithms applied to Test Data generation (and, in particular, to Object-Oriented Evolutionary Testing), and methodologies for automatic refinement (i.e., automating implementation with basis on a formal specification), fault correction (i.e., automatically evolving the input program to make it able to pass a set of Test Cases), improving non-functional criteria (e.g., execution time and power consumption), and reverse engineering (i.e., automatically deriving source code from Bytecode or assembly code).

Although it is possible to generate inputs for certain classes of programs using search-based techniques, the problem of automatically determining whether the corresponding outputs are correct also remains a significant problem; one that is not limited to Evolutionary Testing, but is orthogonal to the entire field of Software Testing. This is because an “oracle” (i.e, a mechanism for checking that the output of a program is correct given some input), is seldom available. Davis and Weyuker [DW81] proposed the use of a “pseudo-oracle” to alleviate this problem. A pseudo-oracle is a program that has been produced to perform the same task as its original counterpart. The two programs, the original and its pseudo-oracle, are run using the same input and their respective outputs compared; any discrepancy may represent a failure on the part of the original program or its pseudo-oracle. Recently, McMinn [McM09] introduced testability transformations (i.e., techniques that change a program in order to make it more “testable”) to automatically generate pseudo-oracles from certain classes of Object-Oriented programs. In [Ton04], the oracle problem is handled by manually adding assertions; Tonella reported that the test suites produced by the Evolutionary Testing method proposed were quite compact, and that augmenting them with assertions would thus be expected to require a minor effort.

Interesting review articles on the topic of SBSE (and Search-Based Software Testing (SBST) in particular) include [McM04, MA05, ABHPW09, HMZ09].

In [McM04], McMinn surveys the use of metaheuristic search techniques for the automatic generation of Test Data. Because the work on SBST had, thus far, been largely restricted to programs of a procedural nature, these are the main subject of this review. Topics include structural and functional testing, the testing of grey-box properties (e.g., safety constraints), and non-functional properties (e.g., worst-case execution time); possible future directions of research for each of individual area are also discussed.

Mantere and Alender [MA05] focus their review on the application of GA-based optimization methods to Software Testing; they stress out that all the researchers in this area report good (or, at least, encouraging) results regarding their use; in fact, if a GA does not seem to outpower Random Search, it is probably poorly implemented. Nevertheless, it is also suggested that, despite being robust, the effectiveness of GAs tends to depend on implementation details (e.g., on how the problem is encoded).

Ali *et. al* present a systematic review on the way SBST techniques have been empirically assessed in [ABHPW09]. Contributions include guidelines on how to conduct empirical studies in SBST, the observation that studies need to, more systematically and rigorously, account for the random variation in the results generated by any metaheuristic algorithm, and the verification that it is impossible to assess how a metaheuristic technique performs in absolute terms – to be able to conclude on its usefulness, a proposed technique needs to be compared with simpler and existing alternatives to determine whether it brings any advantage.

Harman, Mansouri and Zhang [HMZ09] provide a thorough index and classification of SBSE-related literature, supported by an online repository[‡‡] of SBSE publications maintained by Zhang. Local search, Simulated Annealing, GAs and GP are identified as the most widely used optimisation and search techniques; this paper also reveals that nearly two thirds of the overall SBSE literature are concerned with Software Engineering applications relating to Software Testing, with structural Test Data generation being the most studied sub-topic.

3.2 Conclusions

An analysis of the related work on Object-Oriented Evolutionary Testing described above allows drawing some conclusions.

[‡‡]http://www.sebase.org/sbse/publications/

Firstly, nearly all studies have been developed with basis on the program's structure, with the objective being that of attaining a coverage criterion (usually statement or branch); only [CK06b, AY07a] consider the program's specification.

Also, Java is always the programming language of choice for the purposes of implementation and experimentation (with the exception being [XTdHS08]).

Various Test Objects are considered in the experiments described; however, nearly all works (and, in particular, those that do not use custom-made classes) employ container classes (e.g., Stack, BitSet, Vector, TreeMap) as a basis for their studies, and these are typically extracted from the JDK 1.4.

Finally, the frameworks developed are seldom freely available for other researchers to experiment and compare their approaches; the only exception known to the authors is Tonella's eToc tool. As such, comparisons are usually performed against Random Search (e.g., [SG06, WS07, GR08]). As stated in [AY07c], the lack of a common benchmark, which can be used by researchers to test and compare their techniques, makes it difficult to evaluate the performance of a novel technique against existing ones and poses a significant hindrance to the Evolutionary Testing area.

Chapter 4

A Genetic Programming-based Framework for the Evolutionary Testing of Object-Oriented Software

In Evolutionary Testing, metaheuristic search techniques are used to select and produce high-quality Test Data. The focus of this research was put on employing GP for evolving and generating Test Data for the structural Unit Testing of Object-Oriented Java programs.

The decision of addressing the Unit Test generation problem is supported by the observation that most errors are introduced during the unit stage (cf. Section 2.2.1); a tool for automating Unit Testing would greatly improve this – largely informal and often human-dependant – process, and have a direct impact on the quality and reliability of the implemented systems. Structural adequacy criteria was the testing strategy selected, not only because the Unit Testing process is traditionally White-Box oriented (cf. Section 2.2.2), but also because a formal specification of the Test Objects is seldom available. Java was the programming language of choice, both for implementing the *eCrash* Test Data generation tool and for experimentation purposes; this option has to do with the fact that Java is currently the most ubiquitous Object-Oriented language (cf. Section 2.1). Finally, GP was the evolutionary paradigm employed for evolving Test Data; it is arguably the most natural way to evolve Object-Oriented programs (cf. Section 2.3.1), and its characteristics allow automating both the Test Object

analysis and the Test Data generation processes (as will be demonstrated in the remaining of this Chapter).

Significant contributions to the state-of-the-art of the Object-Oriented Evolutionary Testing area were achieved as a result of this research; these will be described in Chapters 5 to 8. The main objectives of the present Chapter are those of providing an overview of Evolutionary Testing methodology proposed, and of presenting the *eCrash* Test Data generation framework for Object-Oriented Java software.

This Chapter is organised as follows. In the next Section, the Evolutionary Testing methodology proposed is overviewed, and the decisions made regarding the development and implementation of the technical approach to the problem are explained. In Section 4.2, the *eCrash* tool, which embodies the technical approach to automatic Test Data generation proposed, is outlined. Section 4.3 details the Test Object analysis stage and, in Section 4.4, the Test Data generation procedure is explained; special emphasis will be put on the techniques utilised for automating these processes. Finally, in Section 4.5, the concepts presented in this Chapter are summarised.

4.1 Methodology Overview

Our approach to Object-Oriented Evolutionary Testing involves encoding potential solutions (i.e., Unit Test Cases) as STGP [Mon95] individuals; STGP is particularly suited for representing and evolving Object-Oriented programs, which may be represented as *Method Call Trees (MCTs)*. A MCT consists of method nodes, each of which represents a method that will later appear in the decoded Test Program; it is rooted, with the root node representing the MUT. In formal terms, it can be defined as follows:

Definition 4.1.1 ([Wap07])**.** A **method call tree** Ψ, defined by the tuple (N_M, E), is an acyclic directed graph, where N_M is the set of nodes representing the test cluster methods, and $E \subseteq N_M \times N_M$ is the set of the edges connecting the method nodes.

With STGP, types are defined *a priori* in the Function Set and define the constraints involved in MCT construction; in other words, the Function Set contains the set of instructions from which the algorithm can choose when building the MCSs that compose Test Programs. This feature enables the initialization process and the various genetic operations to only construct syntactically correct MCTs, thus restraining the search-space to

the set of compilable Test Programs. The Function Set is defined completely automatically based solely on Test Cluster (i.e, the transitive set of classes which are relevant for testing the CUT) information.

In order for a Test Program to be executed, the genotype (i.e, the MCT) must be decoded into the phenotype (i.e., the Test Program); this can be achieved by linearising the tree by means of a depth-first traversal algorithm. The example MCT depicted in the Figure 2.9 on page 32 encodes the Test Program contained on Listing 2.4 on the same page. The MUT is the `search` method of the `Stack` class – which corresponds to the root node of the MCT. The root node's parameters are provided by its children, the `push` method and the `Object` constructor, with the former having it parameters provided by the `peek` method and `Object` constructor, and so on.

The quality of a particular Test Program is related with the CFG nodes of the MUT which are the targets of the evolutionary search at a given stage of the search process. Test Cases that exercise less explored (or unexplored) CFG nodes and paths are favoured, with the objective of attaining the primary goal of the Test Data generation process – finding a set of Test Cases that achieves full structural coverage of the Test Object. Whenever a Test Program exercises a CFG node, that node is marked as "hit"; the search stops when there are no CFG nodes left to be covered or after a predefined number of generations.

Test Program quality evaluation involves the Test Object's instrumentation (i.e., insertion of additional code into the program in order to collect information about program behavior during execution). Instrumentation and CFG analysis are performed statically, before the Test Data generation process takes place; execution flow analysis and fitness evaluation are performed dynamically.

The algorithm depicted in Figure 4.1 summarises the Evolutionary Testing methodology evolutionary approach to automatic Test Program generation proposed. These concepts were implemented into the *eCrash* automated Test Data generation framework.

4.2 Technical Approach

The *eCrash* tool embodies the approach to Evolutionary Testing of Object-Oriented programs proposed; Figure 4.1 provides an overview of this framework and of the way in which its components interoperate. *eCrash* is composed by the following main modules:

Algorithm 4.1: Methodology overview.

```
Data: class under test, test cluster
Result: test set

foreach class under test do
    instrument for structural tracing;
    create control-flow graphs;
    generate function set;
    parameterise evolutionary run;
    foreach method under test do
        initialize weight of control-flow graph nodes;
        initialize constraint selection rankings;
        generate random population;
        repeat
            if generation > 0 then
                adapt constraint selection rankings;
                select individuals for reproduction;
                apply mutation and crossover operators;
                apply object reuse operator;
                generate new population;
            foreach individual do
                linearise method call tree;
                synthesise test program;
                compile and execute test program;
                if test program is feasible then
                    trace control-flow graph nodes hit;
                    if new control-flow graph nodes are hit then
                        mark new control-flow graph nodes as hit;
                        include test program in test set;
                evaluate fitness of individual;
        until stopping criteria are met;
```

- *Test Object Instrumentation (TOI) Module*: executes the tasks of building the CFG and instrumenting the Test Object.
- *Automatic Test Object Analysis (ATOA) Module*: performs the Test Object analysis; it's main tasks are those of defining the Test Cluster, generating the Function Set and parameterising the Test Program generation process.
- *Test Program Generation (TPG) Module*: iteratively evolves potential solutions to the problem with basis on the GP paradigm.
- *Test Program Evaluation and Management (TPEM) Module*: synthesises, executes and evaluates Test Programs dynamically, and selects the Test Programs to be included into the Test Set.

Test Object analysis is performed offline – i.e., before Test Data generation takes place. As a result of this process, the TPEM Module is provided with the instrumented Test Object and the CFGs, which are required for assessing the quality of the generated Test Programs; and the TPG Module is provided the Parameter and Function Files, which contain all the information necessary for the Evolutionary Computation in Java (ECJ) component [Luk09a] of this module to iteratively evolve Test Cases. The outputs of the TPG and TPEM modules include: the Test Set, which may be provided to an external Unit Testing Framework (e.g., jUnit); and several statistics about the Test Data generation process, e.g. the level of coverage attained, the number of Test Programs generated, and the time spent performing the task. The static Test Object analysis process is detailed in Section 4.3, and the dynamic Test Data generation process is described in Section 4.4.

4.3 Test Object Analysis

This Section details the Test Object analysis phase: the following Subsection starts by describing the CFG building and Test Object instrumentation procedures; next, the Test Cluster Definition stage is detailed; and finally, the Function Set generation and Evolutionary Search parameterisation processes are overviewed. The diagram depicted in Figure 4.2 summarises the entire Test Object Analysis process.

4.3.1 Test Object Instrumentation and CFG Creation

In order for the Test Data Generation process to take place, a preliminary analysis of the Test Object must be performed. Specifically, a CFG providing a representation of the MUTs must be created and the CUT must be instrumented. This will allow evaluating the quality of the generated Test Programs: dynamic analysis requires executing each Test Program, and tracing the CFG nodes exercised in order to gather coverage data.

The CFG building, instrumentation and event tracing processes are performed statically with the aid of Sofya*, a Java Bytecode analysis framework that is particularly suited for developing dynamic analysis tools [KDR07]. 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

*http://sofya.unl.edu/

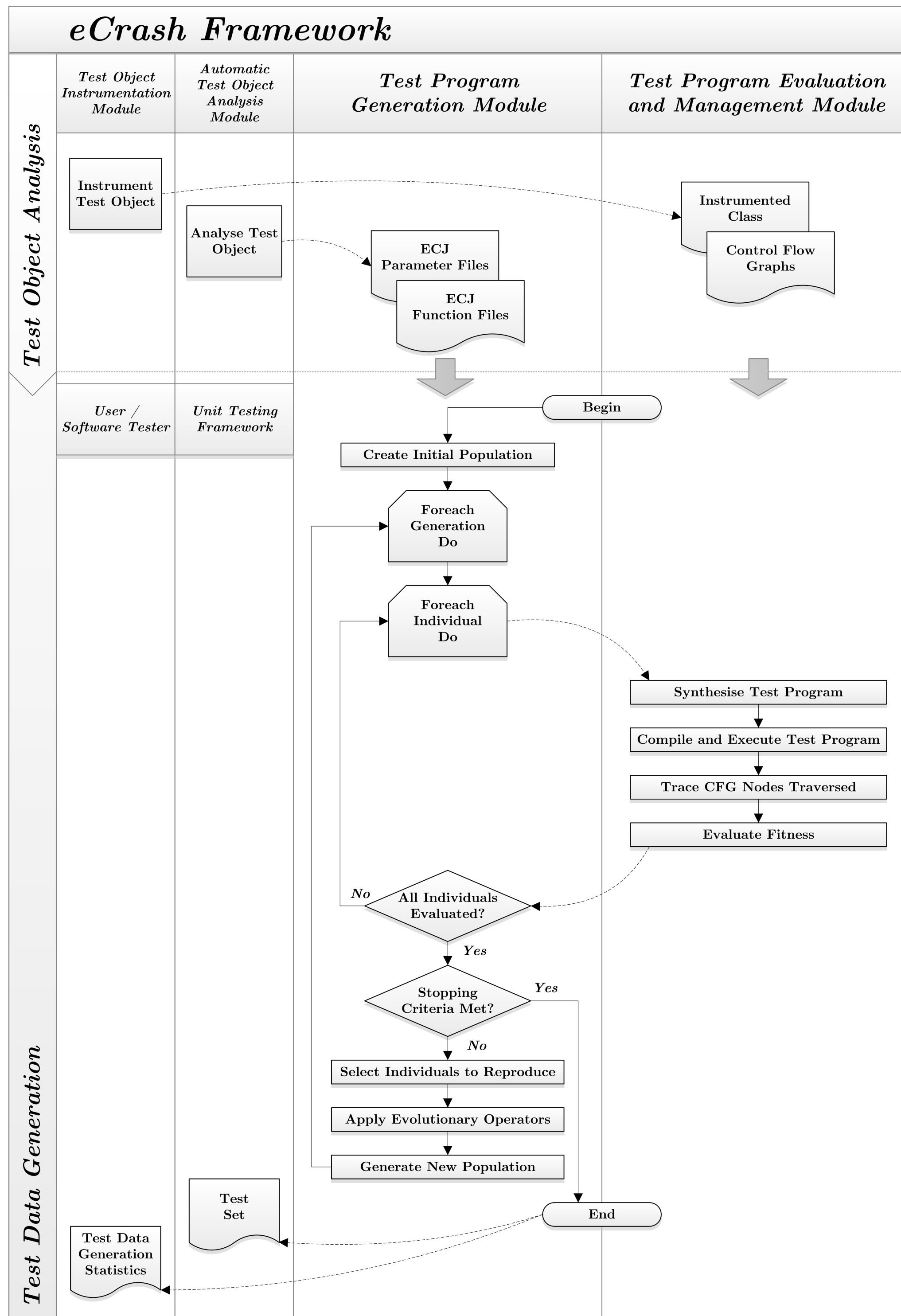

Figure 4.1: Cross-Functional Diagram of the *eCrash* Framework.

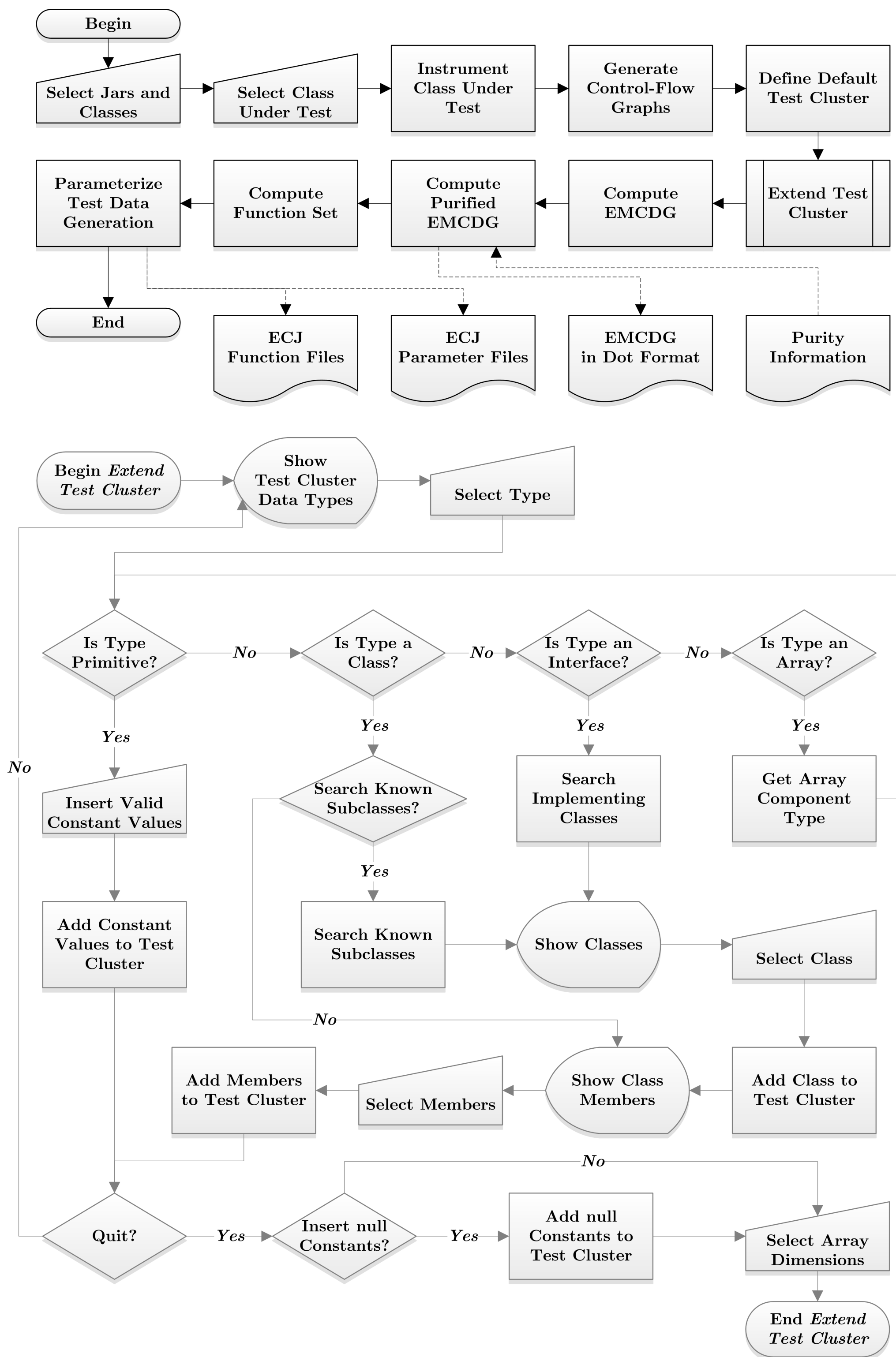

Figure 4.2: Diagram for the Test Object Analysis process.

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 visualise the trace files produced by the executions of instrumented programs.

In the context of the *eCrash* tool, Sofya is employed to instrument classes for structural event dispatch. Basic Block instrumentation enables the observations of the Basic Instruction and Call blocks exercised as a result of the Test Object's execution.

Probe insertion and CFG computation are performed at the Java Bytecode level. 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 components [VDMW06]. Also, 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.

The CFG building procedure involves grouping Bytecode instructions into a smaller set of Basic Instruction blocks and Call blocks, with the intention of simplifying the representation of the Test Object's control flow. *Basic Instruction blocks* encompass regular Bytecode instructions, including the decision and branching instructions that can influence control flow (namely, Bytecode instructions if, goto, jsr, switch, return, ret, throw, sumthrow and exit [LY99]). *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.

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. 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 traversed 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 Bytecode 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.

Let us consider the example CFG depicted in Figure 2.4 on page 17, which was built according to the methodology described above and represents the `search` method of the `Stack` class. Attaining full structural coverage involves traversing the CFG nodes 4, 7, 8 (Basic Instruction blocks) and 2, 5 (Call blocks). Additionally, other types of blocks, which represent virtual operations are defined: an Entry block (block 1), Exit blocks (blocks 9, 10, 11), and Return blocks (blocks 3, 6).

4.3.2 Test Cluster Definition

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; testing a single class thus also involves those classes that appear as parameter types in the signatures CUT's methods. The transitive set of classes which are relevant for testing a particular class is called the Test Cluster for this class.

A call to the `search` method of the `Stack` class (Listing 2.1 on page 17), for example, requires both a `Stack` instance and an `Object` instance to be previously created. As such, the `Object` class must be present in the Test Cluster.

A Test Cluster may be formally defined as follows:

Definition 4.3.1 ([Wap07])**.** Let $\mathcal{C}$ be the set of all classes and $\rhd$ be the class association relation so that $c_i \rhd c_j$ with $c_i, c_j \in \mathcal{C}$ means that class c_i is either a superclass of class c_j, or c_i is associated to c_j by a general association relation. Furthermore, let $\rhd^+$ be the transitive closure of the class association relation $\rhd$. Then,

$$C = \{c_i | c_i \rhd^+ c_t\} \tag{4.1}$$

where $c_i, c_t \in \mathcal{C}$ is the **test cluster** C of the class under test c_t.

The selection of the classes and members which compose the Test Cluster is largely human-dependant. However, the ATOA module of the *eCrash* framework assists the user on this task, by statically examining the data types by means of the Reflection API[†] [ZHR⁺06] and automating all the systematic procedures – e.g., it provides the user with all the concrete classes

[†] http://java.sun.com/javase/6/docs/api/java/lang/reflect/package-summary.html

that instantiate the parameters of interface or abstract types (as suggested by Tonella in [Ton04]). A description of the Test Cluster definition process implemented in *eCrash* follows.

The first task required of eCrash's user is that of selecting one or more jar files[‡] containing all the classes which participating in the analysis; the set of classes to be included in the Test Cluster will be selected among these (and methods, constructors and constants likewise). Also, a non-abstract class must specified as the CUT.

Next, the default Test Cluster is automatically defined; the current methodology for performing this task involves adding the following members to the Test Cluster:

- the CUT;
- all the public methods and constructors of the CUT;
- all the data types (both reference and primitive) appearing as parameter's in the CUT's methods;
- the default constructors (i.e., constructors with no explicit parameters) of all the reference data types, if available;
- the default constant set for each of the primitive Java data types (Table 4.1), which includes acceptable and boundary values and is used to sample the search space in accordance to the methodology proposed in [KJS98].

Table 6.1 on page 85 depicts the default Test Cluster (in accordance to the methodology described above) for the `Stack` class.

After the default Test Cluster has been defined, the user is given the option of extending the default Test Cluster; the Block Diagram for this process is included in Figure 4.2 on page 57. The user may either start by choosing to include any of the classes (and a number of its public methods and constructors) loaded from the jars selected at the beginning, or select a data type already included in the Test Cluster. If the user's choice is the latter, he is provided the following options, depending on the kind of data type selected:

- *Primitive Data Type*: the user is given the option of inserting a valid value;

[‡]Archive of Java classes or libraries.

Primitive Data Type	*Default Constant Set*
boolean	true, false
byte	-1, 0, 1, Byte.MAX_VALUE, Byte.MIN_VALUE
char	'a', '0', '!', Character.MAX_VALUE, Character.MIN_VALUE
double	-0.5d, 0.0d, 0.5d, Double.MAX_VALUE, Double.MIN_VALUE
float	-0.5f, 0.0f, 0.5f, Float.MAX_VALUE, Float.MIN_VALUE
int	-1, 0, 1, Integer.MAX_VALUE, Integer.MIN_VALUE
long	-1L, 0L, 1L, Long.MAX_VALUE, Long.MIN_VALUE
short	-1, 0, 1, Short.MA_VALUE, Short.MIN_VALUE

Table 4.1: Default constant set to be included in the Test Cluster for the primitive Java data types.

- *Class Data Type*: the set of all know subclasses is ascertained by means of Reflection and presented to the user; this set includes all the non-abstract classes, existing in the set of all loaded classes, that extend (directly or transitively) the selected class. Then, the user may select to include any of these classes (and any of its public methods and constructors) in the Test Cluster;

- *Interface Data Type*: the set of all know implementing classes is ascertained by means of Reflection and presented to the user; this set includes all the non-abstract classes, existing in the set of all loaded classes, that implement (directly or transitively) the interface selected. Then, the user may select to include any of these classes (and any of its public methods and constructors) in the Test Cluster;

- *Array Data Type*: the data type of the array component is ascertained by means of Reflection, and depending on the kind of data type (primitive, class or interface), one of the procedures described above follows.

Finally, the user is asked if the `null` constant for reference data types is to be included into the Test Cluster, and required to specify the possible array dimensionalities (if none are provided, the default values 0, 1 and 2 are used).

After the Test Cluster is defined, the ATOA module proceeds to automatically generate the Function Set and, finally, all the configuration files required by ECJ to evolve Test Programs.

4.3.3 Function Set Generation

The Object-Oriented Evolutionary Testing methodology proposed involves encoding and evolving candidate Test Programs as STGP trees; each STGP tree must subscribe to a Function Set which defines the STGP nodes legally permitted in the tree. The Function Set can be generated completely automatically based solely on Test Cluster information, by means of a process which involves an Input Domain Reduction procedure.

The Input Domain Reduction methodology proposed is based on the concept of Purity Analysis [SR04], and is able to prevent the insertion of entries that are irrelevant to the search problem into the Function Set, therefore decreasing the number of distinct Test Programs that can possibly be created while searching for a particular test scenario.

The first task of the Function Set Generation process is that of modelling the call dependencies of the data types and members existing in the Test Cluster; an Extended Method Call Dependence Graph (EMCDG) [WW06b] is employed for this purpose. An EMCDG is a bipartite, directed graph with two types of nodes: *member nodes* represent methods, constructors, or constants; and *data type nodes* represent classes, interfaces, primitive types, and arrays. A link between a member node and a data type node means that the method can only be called if an instance of the linked data type is created in advance; a link between a data type node and a member node means that an instance of the class is created or delivered by the linked member.

The EMCDG can be defined automatically based solely on the Test Cluster information; likewise, the Function Set can be completely derived from the EMCDG. A thorough and illustrated explanation of the whole Function Set Generation process is available in Section 6.2, as part of the description of the Input Domain Reduction strategy proposed. An example EMCDG and the corresponding Function Set for the `search` method of the `Stack` class are shown in Figure 6.1 and Table 6.2 (page 90), respectively.

4.3.4 Evolutionary Search Parameterisation

As was aforementioned, the ECJ framework is utilised by *eCrash*'s Test Program Generation module for evolving potential solutions for a given Evolutionary Testing problem. ECJ is a research package that incorporates several Universal Evolutionary Algorithms, and includes built-in support for STGP. It is highly flexible, having nearly all classes and their settings

being dynamically determined at runtime. Even though parameters can be defined programatically, they rarely are; typically, a hierarchical set of parameter files are defined to set up the Evolutionary Algorithm's configuration, with Function Files being utilised to store the information contained by a particular Function Set entry and, consequently, the data which will be contained in the STGP trees' nodes.

ECJ relies heavily on Parameter and Function files for nearly every conceivable configuration setting[§] and, as such, Function Set information must be parameterised accordingly.

The ATOA module of the *eCrash* framework automates the generation of problem-specific Parameter and Function Files: the former parameterise Function Set information, whereas the latter are built with basis on the corresponding member (constructor, method, or constant) information.

Listings B.1 and B.2 in Appendix B depict, respectively, example Parameter and Function files generated by *eCrash* as a result of the analysis of the `search` method of the `Stack` class.

The Parameter File is divided into 3 sections: General Parameters, Test Object Specific Parameters, and MUT Specific Parameters. The *General Parameters* mostly encompass Evolutionary Algorithms configurations (such as termination criteria, population size, evolutionary operators, selection strategy, and tree builders); these are usually defined and tweaked by the user (e.g., depending on the resources available or in order to experiment with different breeding strategies). The *Test Object Specific Parameters* encompass all problem-related configurations, and include the definition of atomic types, set types, and GP node constraints; these are defined automatically solely with basis on Function Set information. The *MUT Specific Parameters* section is basically used to define the root node of the STGP tree – which must necessarily be the MUT.

Function Files encode all the information that will be included into the MCTs' nodes, which will subsequently be used to decode the STGP tree into the Test Program (by means of the process described in Section 4.4.3). Relevant information includes the type of node (contructor, method, or constant), the member's parameters data types, and the return value data type. Each node constraint defined in the Node Constraints subsection of the Test Object Specific section of the Parameter File must be associated with a distinct Function File.

[§]http://cs.gmu.edu/ eclab/projects/ecj/docs/parameters.html

4.4 Test Data Generation

This Section details the Test Data generation phase: the following Subsection starts by describing the preparatory steps which precede the evolutionary run; in Subsection 4.4.2, the iterative process by means of which candidate solutions to the problem are created is explained; the process of transforming the individuals' genotypes (i.e, the MCTs) into the phenotypes (i.e, the Test Programs) is detailed in Subsection 4.4.3; and Subsection 4.4.4 overviews the methodology utilised for ascertaining the quality of Test Programs and computing the corresponding individuals' fitness. The diagram depicted in Figure 4.3 summarises the whole Test Data Generation process as implemented into the *eCrash* tool.

4.4.1 Setting Up The Evolutionary Run

The main goal of each evolutionary run is to find a set of Test Programs that achieves full structural coverage of the CFG representing a particular MUT; as such, before commencing the evolutionary process, a list of the CFG nodes remaining – which initially includes all the CFG nodes – is created.

At this point, the CFG nodes' weights must also be initialised. As will be described in Section 4.4.4, the issue of steering the search towards the traversal of interesting CFG nodes and paths was addressed by assigning weights to the CFG nodes, which are re-evaluated every generation. At the beginning, all CFG nodes are assigned a predefined weight value, in accordance to the strategy detailed in Chapter 5.

The final step preceding the creation of the initial population of candidate solutions is that of initialising the contraints' selection probabilities. This particular parameter is related with the Adaptive Evolutionary Testing strategy detailed in Chapter 7, which promotes the introduction of relevant instructions into the generated Test Programs by means of Mutation; the instructions from which the GP's tree building algorithm can choose are ranked, with their rankings being updated every generation in accordance to the feedback obtained from the individuals evaluated in the preceding generation. The initial constraints' selection probabilities must be defined in the Parameter Files (e.g., line 119 of Listing B.1); the current strategy is that of defining equal initial selection probabilities to all the constraints.

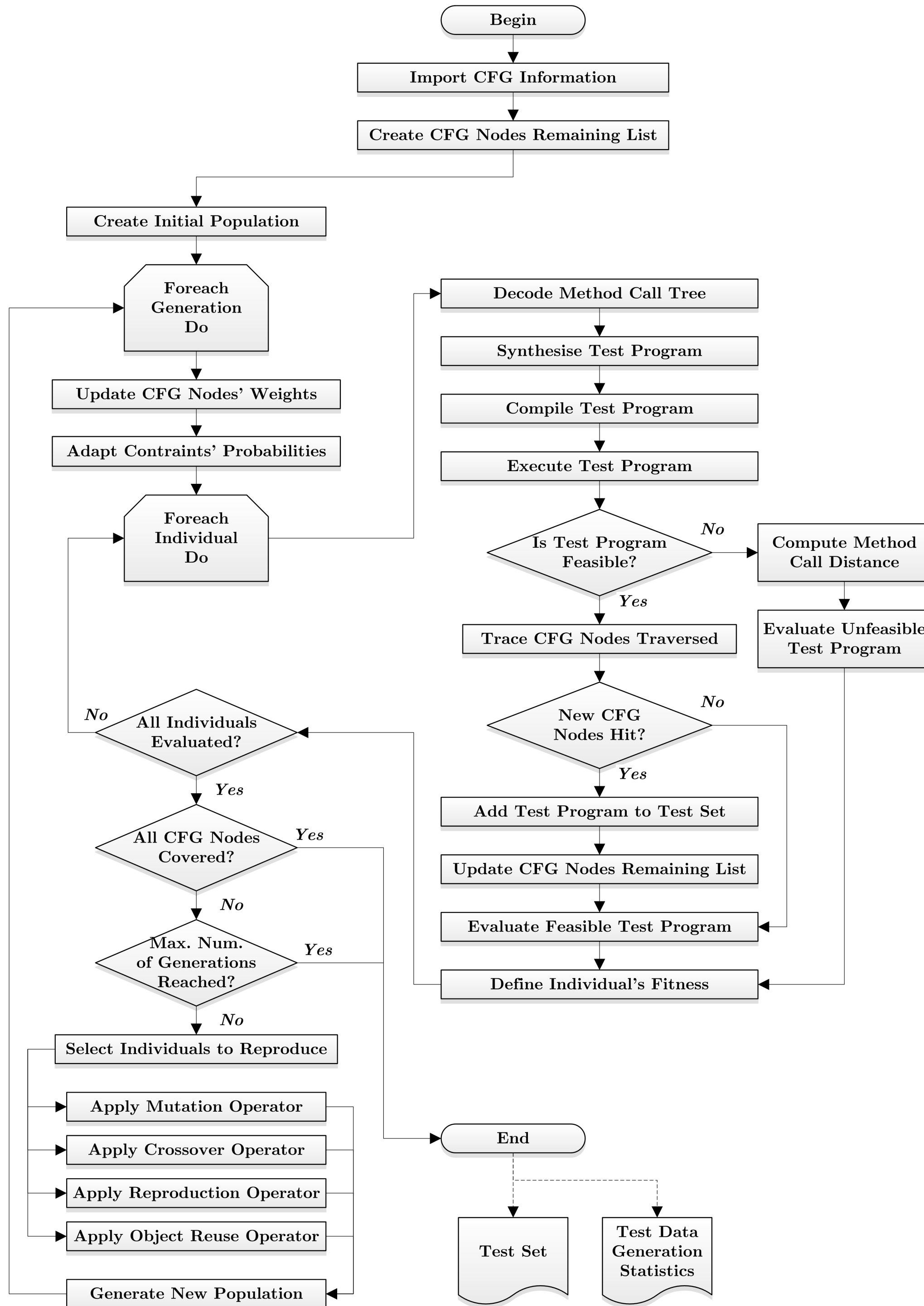

Figure 4.3: Diagram for the Test Data generation process.

4.4.2 Evolving Test Programs

Test Programs are evolved while there are CFG nodes left to be covered, or until a predefined number of generations is reached; the maximum number of generations, and the population size likewise, must have been defined in the Parameter File (e.g., lines 9 and 14 of Listing B.1).

Each individual must be decoded into the corresponding Test Program, compiled and executed, so as to allow verifying if it should be included into the Test Set; whenever a Test Case "hits" an unexercised CFG node, that node is removed from the CFG Nodes Remaining list, and the Test Case is added to the Test Set. Also, the fitness of individuals (of which their probability of being selected for breeding depends) is assessed differently depending on the individuals feasibility; nevertheless, it requires the Test Program's execution.

The initial population is composed by individuals created randomly by means of the selected tree building algorithm (e.g., Full and Grow). Subsequent populations are formed by individuals originated from those existing in the preceding population, which may either be cloned directly (if the Reproduction operator is used for breeding individuals) or altered before being copied (e.g., if the Mutation of Crossover operators are applied). Distinct probabilities of selecting the breeding operators may also be defined.

Individuals may also be bread using the Object Reuse operator, which is an important component of the Object Reuse methodology detailed in Chapter 8. In short, the Object Reuse operator enables the introduction of "At-Nodes" into existing trees; At-Nodes are custom-made STGP nodes that "point to" other nodes, thus effectively enabling the creation of edges to nodes that are already part of the tree, and allowing the reuse of subtrees. The objective is that of permitting a single object instance to be passed to multiple methods as an argument (or multiple times to the same method as arguments), thus enabling the traversal of structural entities in the CFG that would not be reachable otherwise.

4.4.3 Decoding Test Programs

Test Program quality evaluation involves the MUT's instrumentation and posterior execution using the generated Test Programs, with the intention of collecting trace information with which to derive coverage metrics. Decoding an MCT (i.e, the genotype) into the Test Program (i.e., the phenotype) for execution is a two step process which involves: the linearisation of the

MCT, so as to obtain the MCS; and the translation of the MCS into the Test Program. Therefore:

- the MCS corresponds to the internal representation of the Test Program; specifically, it corresponds to the linearised MCT. Listing 2.3 on page 32 depicts a MCS obtained from the linearisation of the MCT depicted in Figure 2.9 (on the same page);
- the Test Program corresponds the syntactically correct, compilable, and executable version of the MCS, according to the programming language of choice (currently, only Java is supported by *eCrash*). Listing 2.4 on page 32 contains the Java Test Program synthesised from the MCS shown in Listing 2.3.

Algorithm 4.2 details the depth-first transversal algorithm utilised to linearise the STGP trees. It should be noted that this algorithm is solely intended to decode those MCTs that do not result from the application of the Object Reuse Operator; the methodology for MCT linearisation in the presence of At-Nodes will be detailed in Chapter 8 as part of the description of the Object Reuse strategy.

The Test Program's source-code synthesis is performed by translating MCSs into Test Programs using the information contained in each MCS entry. Specifically, each MCS entry contains a Method Information Object (MIO), which encloses: the method signature data necessary for the Test Program's source code to be assembled, i.e. the information contained in the Function Files, such as a method's name and class, parameter types and return type (e.g., Listing B.2 in Appendix B); and references to other MIOs providing the parameters (if any) for that method. The Test Program thus corresponds to a syntactically correct translation of the MCS. Algorithm 4.3 depicts the Test Program synthesis procedure.

4.4.4 Evaluating Test Programs

The quality of a particular Test Program 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 CFG nodes and paths are favoured, with the objective of finding a set of Test Cases that achieves full structural coverage of the Test Object.

The issue of steering the search towards the traversal of interesting CFG nodes and paths was addressed by assigning weights to the CFG nodes; the

Algorithm 4.2: Algorithm for Method Call Tree linearisation in the absence of At-Nodes.

```
Data: Method Call Tree
Result: Method Call Sequence

Global Variables:
Current Node ← Root Node;
Previous MIO ← null;
MCS ← empty sequence;

begin Function linearizeMCT(Current Node)
    if Current Node ≠ Root Node then
        Previous MIO ← get MIO from from Parent Node of Current Node;
    Current MIO ← get MIO from Current Node;
    if Previous MIO ≠ null then
        add Current MIO to Parameter Providers List of Previous MIO;
    Child Nodes List ← get Child Nodes List from Current Node;
    foreach Child Node in Child Nodes List do
        call linearizeMCT(Child Node);
    add Current MIO to MCS;
```

higher the weight of a given node, the higher the cost of exercising it, and hence the higher the cost of traversing the corresponding control-flow path. Additionally, the weights of CFG nodes are re-evaluated at the beginning of every generation; nodes which have been recurrently traversed in previous generations and/or lead to uninteresting paths are penalised.

At the beginning of each generation the weight of each CFG node is multiplied by: a *weight decrease constant* value α, so as to decrease the weight of all CFG nodes indiscriminately; a *hit count factor*, which worsens the weight of recurrently hit CFG nodes; and a *path factor*, which improves the weight of nodes that lead to interesting nodes and belong to interesting paths. For feasible Test Cases, the fitness is evaluated as being the average weight of the nodes exercised. This computation is performed with basis on the trace information; relevant data includes the "Hit List" – i.e. the set of traversed CFG nodes. For unfeasible Test Cases, the fitness of the individual is calculated in terms of the distance between the index of the method call that threw the exception and the MCS's length; the higher the percentage of instructions executed, the higher the individual's quality. Also, an *unfeasible penalty constant* value β is added to the final fitness value, so as to penalise unfeasibility.

The Test Program evaluation strategy utilised will be thoroughly described in Chapter 5.

Algorithm 4.3: Algorithm for Test Program synthesis with basis on the Method Call Sequence.

```
Data: Method Call Sequence
Result: Test Program
counter ← 1;
variableName ← null;
foreach MIO in MCS do
    variableName = dataTypeName + counter ;
    if MIO contains a CONSTANT then
        i ← dataTypeName + " " + variableName + " = " + constant + ";";
    else if MIO contains a CONSTRUCTOR then
        i ← dataTypeName + " " + variableName + " = new " +
        dataTypeName + "(" + providersList.associatedItemInfos + ");";
    else if MIO contains a METHOD then
        if method returns an array then
            i ← dataTypeName + "[] " + variableName + " = " +
            providersList[0].associatedItemInfo + "." + methodName + "(" +
            providersList[1..SIZE-1].associatedItemInfos + ");";
        else if method returns void then
            i ← providersList[0] + "." + methodName + "(" +
            providersList[1..SIZE-1].associatedItemInfos + ");";
        else if method returns a primitive or reference data type then
            i ← dataTypeName + " " + variableName + " = " +
            providersList[0] + "." + methodName + "(" +
            providersList[1..SIZE-1].associatedItemInfos + ");";
    else if MIO contains an ARRAY then
        i ← dataTypeName + "[]" + variableName + " = []{" +
        providersList + "};";
    if associatedItem is RE then
        associatedItemInfo ← variableName;
    else if associatedItem is IP then
        associatedItemInfo ← providersList[0].associatedItemInfo;
    else if associatedItem is Pρ, ρ ∈ [1..SIZE − 1] then
        associatedItemInfo ← providersList[ρ].associatedItemInfo;
    counter ← counter + 1;
    testProgram ← testProgram + i + LINEBREAK;
```

4.5 Summary

This Chapter described a technical approach for automating the generation of Unit Test Data for Object-Oriented software. The concepts presented were implemented into the *eCrash* tool.

eCrash is a Java-based prototype Test Data generation tool; it was developed during the course of the research presented in this Thesis to support the integration of research steps. Achieving the highest level of automation possible was a primary concern, and always underlied its development and implementation; as was mentioned in Section 2.2, the lack of automation is one of the major problems faced by Software Testers today, and a major hindrance which still prevents Object-Oriented software from being adequately tested and validated.

The Object-Oriented Evolutionary Testing approach proposed involves encoding and evolving potential solutions to the problem (i.e., Test Programs) as STGP trees. When evaluating Test Programs, whenever a CFG node is exercised, that node is removed from the Remaining Nodes list, and the Test Program is added to the Test Set; the search stops when there are no CFG nodes left to be covered or after a predefined number of generations. For representing and evolving Test Data, the ECJ package [Luk09a] is used.

Test Object instrumentation and CFG generation are performed statically with the aid of the Sofya framework [KDR07]. The definition of the Function Set must also precede the Test Data generation phase, as each MCT subscribes to a Function Set which defines the STGP nodes legally permitted in the tree. The Function Set is computed automatically with basis on the Test Cluster.

In the following Chapters, significant contributions for enhancing both the efficiency and the effectiveness of Object-Evolutionary Testing approaches will be thoroughly described; additional information on the procedures described in the preceding Sections will also be provided.

Chapter 5

A Strategy for Evaluating Test Programs for the Evolutionary Testing of Object-Oriented Software

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.

With the evaluation methodology proposed, the quality of a particular Test Program 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 Programs that exercise less explored (or unexplored) CFG nodes and paths must be favoured.

However, the execution of Test Programs may abort prematurely if a runtime exception is thrown during execution (cf. Section 2.4.1). When this happens, it is not possible to trace the structural entities transversed in the MUT because the final instruction of the Test Program is not reached. Test Programs that fall into this class are referred to as unfeasible Test Programs – as opposed to feasible Test Programs, which are effectively executed and terminate with a call to the MUT.

As a general rule, longer and more intricate Test Programs are more prone to throw runtime exceptions; however, complex MCSs are often needed

for defining elaborate state scenarios and transversing certain problem nodes [SAY07]. If unfeasible sequences are blindly penalised, the definition of complex Test Cases will be discouraged.

This Chapter presents a strategy for evaluating the quality of both feasible and unfeasible Test Programs. With the proposed approach, unfeasible Test Cases are considered at certain stages of the evolutionary search thus promoting diversity and enhancing the possibility of achieving full coverage.

The following Section starts by describing the CFG node's reevaluation strategy. Sections 5.2 and 5.3 explain how the fitness of feasible and unfeasible Test Programs, respectively, are calculated. The experimental studies performed with the objective of validating the Test Program Evaluation methodology proposed are detailed and discussed in Section 5.4. Finally, Section 5.5 summarises the concepts addressed in this Chapter.

5.1 CFG Nodes' Weights Reevaluation

The issue of steering the search towards the traversal of interesting CFG nodes and paths was address by assigning weights to the CFG nodes; the higher the weight of a given node the higher the cost of exercising it, and hence the higher the cost of transversing the corresponding control-flow path.

Let each CFG node $n \in N$ represent a linear sequence of computations of the MUT. Then, 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 traversing 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}.

The CFG nodes' weights are reevaluated every generation according to Equation 5.1.

$$W_{ni} = (\alpha W_{ni}) \left(\frac{hitC_{ni}}{|T|} + 1 \right) \left(\frac{\sum_{x \in N_s^{ni}} W_x}{|N_s^{ni}| \times \frac{W_{init}}{2}} \right) \tag{5.1}$$

The $hitC_{ni}$ parameter is the "Hit Count", and contains the number of times a particular CFG node was exercised by the Test Programs of the previous generation. T represents the set of Test Cases produced in the previous generation, with $|T|$ being its cardinality.

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

In summary, each generation the weight of a given node is multiplied by:

- the *weight decrease constant* value α, so as to decrease the weight of all CFG nodes indiscriminately;
- the *hit count factor*, which worsens the weight of recurrently hit CFG nodes;
- the *path factor*, which improves the weight of nodes that lead to interesting nodes and belong to interesting paths.

After being reevaluated, the weights of all the nodes are normalised to the nodes' initial weight W_{ni} in accordance to Equation 5.2.

$$W_{ni} = \frac{W_{ni} \times W_{init}}{W_{max}} \tag{5.2}$$

W_{max} corresponds to the maximum value for the weight existing in N.

5.2 Evaluation of Feasible Test Cases

For feasible Test Cases, the fitness is computed with basis on their trace information; relevant trace information includes the "Hit List" – i.e., the set $H_t, H_t \subseteq N$ of traversed CFG nodes. The fitness of feasible Test Cases is, thus, evaluated as follows:

$$Fitness_{feasible}(t) = \frac{\sum_{h \in H_t} W_h}{|H_t|} \tag{5.3}$$

This strategy causes the fitness of feasible Test Programs that exercise recurrently traversed structures to fluctuate throughout the search process; frequently hit nodes will have their weight increased, thus worsening the fitness of the Test Cases that exercise them.

5.3 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 MCS length $seqLen_t$.

Also, an *unfeasible penalty constant* value β is added to the final fitness value, so as to penalise unfeasibility.

$$Fitness_{unfeasible}(t) = \beta + \frac{(seqLen_t - exInd_t) \times 100}{seqLen_t} \quad (5.4)$$

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 MCSs.

5.4 Experimental Studies

In this Section, the empirical studies implemented with the objectives of validating and observing the impact of our Test Program Evaluation strategy are described and discussed.

The Java `Stack` and `BitSet` classes (JDK 1.4) were used as Test Objects. The rationale for employing these classes is related with the fact that they represent "real-world" problems and, being container classes, possess the interesting property of containing explicit state, which is only controlled through a series of method calls [AY07b]. Additionally, they have been used in several other case studies described in literature (cf. Chapter 3), providing an adequate testbed in the lack of common benchmark cluster that can be used to test and compare different techniques.

The main objectives of these experiments were those of:

- analysing the impact of distinct configurations for the *probabilities of the evolutionary operators* Mutation, Reproduction and Crossover;
- assessing the effect of setting different values for the *Test Case evaluation parameters* – the *weight decrease constant* α (Equation 5.1) and the *unfeasible penalty constant* β (Equation 5.4).

For evolving Test Cases, ECJ was configured using a single population of 5 individuals. The MUTs' CFG nodes were initialised with a weight W_{ni} of 200. The search stopped if an ideal individual was found or after 200 generations. For the generation of individuals a multi-breeding pipeline was used, 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:

- a *Reproduction pipeline*, which simply makes a copy of the individuals it receives from its source;

- a *Crossover pipeline*, which performs a strongly-typed version of Subtree Crossover [Koz92] – two individuals are selected, a single tree is chosen in each such that the two trees have the same constraints, a random node is chosen in each tree such that the two nodes have the same return type, and finally the swap is performed;

- and a *Mutation pipeline*, which implements a strongly-typed version of Point Mutation [Koz92] – an individual is selected, a random node is selected, and the subtree rooted at that node is replaced by a new valid tree.

The selection method employed was *Tournament Selection* with a size of 2, which means that first 2 individuals are chosen at random from the population, and then the one with the best fitness is selected.

5.4.1 Probabilities of Operators Study

This particular experiment was performed with the intention of assessing the impact of the evolutionary operators' probabilities on the Test Case generation process. In order to do so, 4 distinct parametrizations of the multi-breeding pipeline were defined, having:

1. a high probability of selecting the Mutation pipeline;

2. a high probability of selecting the Crossover pipeline;

3. a high probability of selecting the Reproduction pipeline;

4. equal probabilities of selecting either pipeline.

For each of the above multi-breeding pipeline parametrisations, 20 runs were executed for the `Stack`'s methods, and 10 runs were executed for the `BitSet`'s methods.

The *weight decrease constant* α was set to 0.9, and the unfeasible penalty constant β was defined as 150. It should be noted that the definition of these values was heuristic, as no experiments had been performed that allowed a fundamented choice; these were conducted later, and are described in the following Section.

Table 5.1 summarises the results obtained. The statistics show that the strategy of assigning balanced selection probabilities (Reproduction: 0.33, Crossover: 0.33, Mutation: 0.34) to the available breeding pipelines yields the best results. For the `Stack` class, this configuration was the only one in which full coverage was achieved in all of the runs (in, at most, 200 generations), and it was also the best in terms of the average number of generations required to attain it; for the `BitSet` class, it was the only configuration in which full coverage was attained at least once for all the MUTs. The worst results were obtained using the parametrisation in which the reproduction breeding pipeline was given the highest probability of selection.

5.4.2 Evaluation Parameters Study

In this case study, different combinations of values for the α and β parameters were studied, with the intention of analysing the impact of the Test Program evaluation parameters on the evolutionary search. 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 experiment described in Subsection 5.4.1 – i.e., the probabilities for Reproduction, Crossover and Mutation were set to 0.33, 0.33 and 0.34, respectively. The other configurations remained unaltered. All the 9 combinations of the α and β values were employed; 20 runs were executed for each of the `Stack`'s MUTs, and 5 runs where executed for the `BitSet`'s methods.

The results obtained are summarised in Table 5.2. The statistics clearly show that the best configuration for the Test Case evaluation parameters is that of assigning a value of 150 to β and a value of 0.5 to α.

5.4.3 Discussion

Search-based Test Data Generation is a challenging research topic; 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 the approach proposed, the evaluation parameters α and

	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:0.3 c:0.3 m:0.3	
	#g	%f	#g	%f	#g	%f	#g	%f
Stack								
empty	10,2	100%	11,2	100%	17,5	100%	4,5	100%
peek	6,6	100%	10,7	100%	9,4	100%	2,8	100%
pop	6,5	100%	8,9	100%	8,6	100%	2,8	100%
push	20,6	100%	16,4	57%	37,2	95%	2,5	100%
search	48,9	95%	48,2	57%	98,8	82%	18,7	100%
BitSet								
hashCode	1,4	100%	2,0	100%	1,3	100%	1,4	100%
clear(int)	65,3	100%	154,8	40%	140,8	50%	92,1	90%
clear()	7,2	100%	31,2	100%	28,6	100%	12,6	100%
clear(int,int)	181,5	20%	-	0%	-	0%	175,4	30%
toString	7,6	100%	29,6	100%	32,0	100%	8,6	100%
isEmpty	7,6	100%	52,4	100%	32,0	100%	8,6	100%
length	41,4	100%	125,4	60%	126,2	70%	49,2	100%
get(int,int)	-	0%	-	0%	-	0%	186,2	30%
get(int)	57,8	100%	184,8	30%	137,4	80%	75,8	100%
size	1,2	100%	2,0	100%	1,2	100%	1,2	100%
set(int,boolean)	24,8	100%	29,0	100%	37,8	100%	23,6	100%
set(int,int)	42,8	100%	122,2	100%	109,0	100%	44,8	100%
set(int,int,boolean)	32,2	100%	68,2	100%	97,4	80%	25,0	100%
set(int)	37,2	100%	127,0	80%	86,8	100%	51,8	100%
flip(int,int)	80,4	80%	184,8	60%	165,0	40%	89,8	100%
flip(int)	73,4	100%	174,0	40%	148,6	60%	77,8	100%
andNot	38,0	100%	-	0%	104,8	100%	20,8	100%
cardinality	7,6	100%	52,4	100%	32,0	100%	8,6	100%
intersects	127,8	60%	-	0%	150,6	50%	107,4	60%
nextSetBit	105,8	100%	192,2	20%	192,8	20%	114,6	60%
xor	80,6	80%	123,6	50%	133,4	40%	70,3	90%
Averages								
Stack	18,6	99%	19,1	83%	34,3	95%	6,3	100%
BitSet	51,1	80%	97,4	56%	92,5	65%	59,3	81%

Table 5.1: Experimental results for the Probabilities of Operators study: percentage of runs attaining full structural coverage ($\%f$); and average number of generations required to attain full structural coverage ($\#g$); using different combinations for the probabilities of choosing the Reproduction (r), Crossover (c) and Mutation (m) operators.

β	0						150						300					
α	**0.1**		**0.5**		**0.9**		**0.1**		**0.5**		**0.9**		**0.1**		**0.5**		**0.9**	
	%f	#g	%f	#g	%f	#g	%f	#g	%f	#g	%f	#g	%f	#g	%f	#g	%f	#g
`Stack`																		
empty	100%	5	100%	6	100%	5	100%	5	100%	5	100%	5	100%	5	100%	5	100%	5
peek	100%	3	100%	4	100%	3	100%	3	100%	3	100%	3	100%	3	100%	3	100%	3
pop	100%	3	100%	3	100%	3	100%	2	100%	2	100%	3	100%	3	100%	3	100%	3
push	100%	5	100%	5	100%	5	100%	5	100%	5	100%	3	100%	5	100%	5	100%	5
search	100%	18	100%	18	100%	22	100%	16	100%	16	100%	19	100%	16	100%	21	100%	22
`BitSet`																		
hashCode	100%	2	100%	2	100%	2	100%	2	100%	2	100%	2	100%	2	100%	2	100%	2
clear(int)	20%	163	80%	133	80%	107	40%	126	100%	123	100%	105	40%	126	100%	122	80%	135
clear()	100%	15	100%	8	100%	8	100%	13	100%	8	100%	8	100%	7	100%	8	100%	8
clear(int,int)	0%	-	0%	-	0%	-	0%	-	40%	177	0%	-	0%	-	40%	180	20%	181
toString	100%	14	100%	8	100%	8	100%	13	100%	8	100%	8	100%	8	100%	8	100%	8
isEmpty	100%	10	100%	8	100%	8	100%	10	100%	8	100%	8	100%	10	100%	8	100%	8
length	80%	88	100%	59	100%	49	100%	106	100%	67	100%	49	100%	109	100%	44	100%	49
get(int,int)	0%	-	0%	-	0%	-	0%	-	20%	188	0%	-	0%	-	0%	-	0%	-
get(int)	60%	136	100%	97	100%	87	100%	96	100%	83	100%	96	60%	146	100%	86	100%	70
size	100%	1	100%	1	100%	1	100%	1	100%	1	100%	1	100%	1	100%	1	100%	1
set(int,boolean)	100%	73	100%	15	100%	17	100%	73	100%	15	100%	17	100%	73	100%	15	100%	17
set(int,int)	60%	130	100%	40	100%	66	60%	122	100%	45	100%	60	60%	129	100%	45	100%	54
set(int,int,bool)	100%	63	100%	25	100%	24	100%	47	100%	25	100%	27	100%	44	100%	25	100%	27
set(int)	80%	106	100%	68	100%	42	60%	108	100%	62	100%	60	60%	113	100%	62	80%	63
flip(int,int)	60%	145	100%	66	80%	111	60%	122	100%	94	100%	94	80%	136	100%	70	100%	95
flip(int)	20%	178	60%	144	80%	126	40%	170	100%	102	100%	79	60%	139	100%	97	100%	82
andNot	40%	137	100%	21	100%	47	60%	147	100%	21	100%	47	80%	125	100%	21	100%	47
cardinality	100%	15	100%	8	100%	8	100%	9	100%	8	100%	8	100%	16	100%	8	100%	8
intersects	0%	-	40%	149	60%	171	20%	173	60%	149	60%	149	0%	-	20%	190	60%	180
nextSetBit	20%	166	80%	118	60%	140	40%	158	60%	132	60%	132	40%	172	60%	145	60%	139
xor	0,8	124	100%	34	100%	51	80%	92	100%	26	100%	37	60%	89	100%	33	100%	37
Averages																		
`Stack`	100%	7	100%	7	100%	8	100%	6	100%	6	100%	6	100%	6	100%	7	100%	8
`BitSet`	63%	87	84%	53	84%	57	70%	84	90%	64	87%	52	69%	80	87%	59	86%	60

Table 5.2: Experimental results for the Evaluation Parameters study: percentage of runs attaining full structural coverage ($\%f$); and average number of generations required to attain full structural coverage ($\#g$); using different combinations for the α and β parameters.

β and the evolutionary operators' selection probabilities play a central role in the Test Program 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 traversal of unexercised structures 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 Programs 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 experiments performed allow drawing a preliminary conclusion: the assumption made on the Probabilities of Operators case study (Subsection 5.4.1), in which $\alpha = 0.9$ was employed as being an adequate value, was incorrect. Using a value of 0.5 for this evaluation parameter yielded better results.

On the other hand, it is possible to affirm that the strategy of assigning the value 150 to the *unfeasible penalty constant* β yields 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 normalised each generation (Equation 5.2). 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 (Equation 5.3).

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 unfeasible 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.

5.5 Summary

The state problem of Object-Oriented programs requires the definition of carefully fine-tuned methodologies that promote the transversal of problematic structures and difficult control-flow paths. We proposed tackling this particular challenge by defining weighted CFG nodes. The direction of the search is under constant adaptation, as the weight of CFG nodes is dynamically reevaluated every generation. Also, the fitness of feasible Test Cases fluctuates 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 because they exercise recurrently traversed structures. In conjunction with the impact of the evolutionary operators, a good compromise between the intensification and diversification of the search can be achieved.

Chapter 6

Employing Purity Analysis for Reducing the Input Domain of Object-Oriented Evolutionary Testing Problems

Object-Oriented Evolutionary Testing problems are hindered by the explosive size of the search space, which encompasses the arguments to the implicit and explicit parameters of the Test Object's public methods. Recent surveys [HHL+07] indicate that strategies for reducing the input domain can greatly increase the performance of this category of search problems.

Input Domain Reduction deals with the removal of irrelevant variables, with the intention of decreasing the number of distinct Test Programs that can possibly be created while searching for a particular test scenario; in this Chapter, a strategy for employing Purity Analysis [SR04, SR05, XPV07] as a means to reduce the input domain of Object-Oriented programs is described.

Our Evolutionary Testing approach involves representing Test Programs using the STGP paradigm; Purity Analysis is particularly useful in this context because it provides a means to automatically identify and remove Function Set entries that do not contribute to the definition of interesting test scenarios.

This Chapter is organised as follows. In the next Section, background on Purity Analysis and Input Domain Reduction is provided, and related

work is contextualised. The Input Domain Reduction methodology, which involves the generation of a purified Function Set, is detailed in Section 6.2. Experimental Studies are presented discussed in Section 6.3, and the final Section summarises the main achievements attained.

6.1 Purity Analysis

Methods in Object-Oriented languages often modify the objects that they access, including the implicit and explicit parameters. However, some methods have no externally visible side effects when executed or, at least – and according to the particular definition – the extent of these side effects is limited in some way [XPV07]; these are called *pure methods*. The knowledge that a method is pure or has no externally visible side effects is especially important because it guarantees that invocations of the method will not interfere with other computations.

According to the definition provided by the JML, a pure method is one which does not [LBR06]:

- perform Input/Output operations;
- write to any pre-existing objects;
- or invoke any impure methods.

This definition allows a method to change the state of newly allocated objects and/or construct objects and return them as a result.

JML is annotation based, requiring purity information to be provided manually by users. Salcianu and Rinard have, however, presented a systematic Purity Analysis methodology [SR05] based on a previous points-to and escape analysis [WR99]. Their purity definition is similar to the one specified by JML: a pure method can read from or write to local objects, and can also create, modify and return new objects not present in the input state.

More interestingly, their methodology is able to identify important purity properties even when a method is not pure. Specifically, Salcianu and Rinard's approach is able to recognise safe and read-only parameters:

- a parameter is *read-only* if the method does not write the parameter or any objects reachable from the parameter;

- a parameter is *safe* if it is read-only, and the method does not create any new externally visible paths in the heap to objects reachable from the parameter.

Those parameters that do not fall into these categories are called *read-/write*.

This type of analysis on the purity of method parameters will henceforth be referred to as *Parameter Purity Analysis*.

Purity Analysis – and Parameter Purity Analysis in particular – is especially useful in the context of Object-Oriented Evolutionary Testing, as it provides a means to automatically identify and remove entries that are irrelevant to the search problem, reducing the size of the set of method calls from which the algorithm can choose when constructing the MCTs that encode Test Programs. The following example illustrates this statement.

Listing 2.4 on page 32 depicts a sample Test Program generated automatically by the *eCrash* tool without using Purity Analysis. The MUT is the `search` method of the `Stack` class; instructions 1, 3 and 5 instantiate new objects, whereas instructions 2 and 4 aim to change the state of the `stack0` instance variable that will be used, as the implicit parameter, in the call to the MUT at instruction 6. However, instruction 2 does not actually change the state of the `stack0` instance: the `peek` method simply looks at the object at the top of the stack without removing it and without changing the state of the stack. This instruction does not interfere with other computations and could, therefore, be safely eliminated without disabling the possibility of traversing any specific structure in the MUT.

What's more, irrelevant instructions may render the Test Program unfeasible by throwing runtime exceptions during execution; the Test Program mentioned above, for example, throws an `EmptyStackException` at instruction 2.

Performing Parameter Purity Analysis on the implicit parameters of the `peek` method would allow marking it as being *safe*. Therefore, this particular method could be discarded of being a `Stack` data type providers, and instructions 2 could be excluded from the set of instructions selectable by the Test Program generation algorithm.

6.1.1 Related Work

There has been little investigation of the relationship between the size of the input domain (i.e., the search space) and performance of search-based

algorithms; Harman *et al.* [HHL+07] were the first to characterise and empirically explore the search-space/search-algorithm relationship for search-based Test Data generation. In this work, static analysis was used to remove irrelevant variables for a given Test Data generation problem (i.e., from the set of possible input vector parameter-value combinations), thereby reducing the search space size. However, this study focused on procedural software and primitive parameter values; to the best of our knowledge, only two works addressed the issue of reducing the input domain of Object-Oriented Evolutionary Testing problems.

In [AY07b], Arcuri and Yao presented a way to reduce the search space for Object-Oriented software by eliminating the functions that cannot give any further help to the search, so as to avoid inserting method calls that do not change the state of the object in the MCS. For determining if a function is read-only, a syntactic analysis of the source code is performed. Additionally, and because only container classes were used in the experiments, a database of common read-only function names (e.g., `insert`, `add`, `push`) was built and used to eliminate such functions using string matching algorithms. For the container classes employed in the experiments, an improvement of 65.5% (on average) in terms of efficiency was reported .

In Arjan Seesing's Master Thesis report [SG06], Purity Analysis was proposed as a means to improve the performance of a search-based approach to Test Data generation for Object-Oriented software. A GP approach was employed for creating test software for Object-Oriented systems, and Purity Analysis was integrated into the test tool described (*EvoTest*). Its usage is reported to almost double the coverage/time performance of the tool. However, the methodology lacked complete automation; it was stated that the analysis performed by the *EvoTest* tool still made many mistakes, and manual annotations were allowed and used to complement the information generated automatically. Additionally, the usage of Parameter Purity Analysis is not reported. Also, because Input Domain Reduction was not the primary focus of this work, the procedure is not thoroughly explored and described.

The Input Domain Reduction strategy presented in this Chapter builds on the concept of Purity Analysis; however, the methodology proposed is systematic and fully automated. What's more, we introduce the usage of Parameter Purity Analysis, which allows the automatic identification and removal of entries even if the corresponding methods are not entirely pure.

6.2 Purified Function Set Generation

This Section describes the Input Domain Reduction strategy proposed; Algorithm 6.1 summarises the process. The output is the purified Function Set, which specifies the nodes legally permitted in the STGP trees that model Test Programs.

Algorithm 6.1: Input Domain Reduction strategy overview.

```
Data: class under test
Result: purified function set
define test cluster;
foreach method in test cluster do
    foreach parameter do
        annotate parameter purity;
    compute types required table;
    compute provided table;
initialize EMCDG with nodes;
connect EMCDG nodes with edges;
remove irrelevant edges;
create purified EMCDG;
create purified function set;
```

The first task of the purified Function Set generation process is that of loading the user provided CUT; the `Stack` class will be used throughout this Section for illustration purposes.

Next, the Test Cluster must be defined. A thorough description of the Test Cluster definition phase of the Test Object Analysis process is provided in Section 4.3.2 on page 59. For the purpose of this example, the Test Cluster depicted in Table 6.1 will be considered.

Test Cluster

Data Types	*Members*
Object	Object()
Stack	Stack()
	Object pop()
	Object push(Object)
	boolean empty()
	Object peek()

Table 6.1: Example Test Cluster for the `Stack` Class.

Parameter Purity Analysis is performed on the parameters (both implicit and explicit) of the methods included in the Test Cluster with the aid of

Parameter Purity Analysis Results

Method	*Parameter Purity*
Object pop()	IP is read/write
Object push(Object)	IP is read/write
	P0 is read-only
boolean empty()	IP is safe
Object peek()	IP is safe
int search(Object)	IP is safe
	P0 is safe

Table 6.2: Parameter Purity Analysis results for the `Stack` class.

the Soot Java Optimization Framework [VRCG+99]. Purity Analysis was implemented into the Soot framework by Antoine Mine, in conformity to the methodology proposed by Salcianu and Rinard [SR04, SR05]. Parameters are annotated as being *safe*, *read-only* or *read/write*.

The Parameter Purity Analysis results for the methods of the aforementioned Test Cluster are shown in Table 6.2; IP refers to the Implicit Parameter, and P0 refers to the first Explicit Parameter.

Next, two tables are computed:

- a *Data Types Required Table*, which identifies the data types required to build a method call for each member (i.e., the parameter data types);
- and a *Data Types Provided Table*, which identifies the data types potentially supplied by each member (i.e., the parameter reference data types and the return types).

The Data Types Required and Data Types Provided tables for the Test Cluster under consideration are depicted in Tables 6.3 and 6.4, respectively.

These tables are built with basis on the methods' signatures and return type information, and the entries are labelled with information on the *Associated Item*. The Associated Item label links data types to the implicit parameter, to an explicit parameter, or to the return value; it allows the unambiguous definition of the data type's provider/consumer. Without this information, it would not be possible to construct the instructions in the posterior Test Data Generation phase.

The list of possible labels for the *Associated Item* is the following:

Data Types Required Table

Member	*Data Types Required*
Stack()	-
Object pop()	Stack [IP]
Object push(Object)	Stack [IP]
	Object [P0]
boolean empty()	Stack [IP]
Object peek()	Stack [IP]
int search(Object)	Stack [IP]
	Object [P0]

Table 6.3: Data Types required by the Public Members of the `Stack` class.

Data Types Provided Table

Data Type	*Provider Members*
Stack	Stack() [IP]
	Object pop() [IP]
	Object push(Object) [IP]
	boolean empty() [IP]
	Object peek() [IP]
	int search(Object) [IP]
Object	Object() [IP]
	Object pop() [RE]
	Object push(Object) [RE, P0]
	Object peek() [RE]
	int search(Object) [P0]

Table 6.4: Data Types provided by the Public Members of the `Stack` class.

- *Implicit Parameter (IP)* – the data type is associated to the implicit parameter;
- *Explicit Parameter n (Pn)* – the data type is associated to the explicit parameter number $n \in \mathbb{N}_0$;
- *Return Value (RE)* – the data type is associated to the return value.

At this point, all the data required for building the EMCDG and modelling call dependencies has been assembled.

The EMCDG is initialised, in accordance to the information contained in the Test Cluster, with two types of nodes: *data type nodes* and *member nodes*. The EMCDG nodes are then connected, in accordance to the infor-

mation contained in the Data Types Required and Data Types Provided tables:

- a directed edge between a data type (origin) and a member (destination) means that the data type at the origin is provided by the member at the destination;

- a direct edge between a member (origin) and a data type (destination) means that the member at the origin requires the data type at the destination.

Edge information is complemented with a label containing information on the Associated Item, in order to complete the EMCDG definition. Algorithm 6.2 details the EMCDG generation process.

The *purified EMCDG* is computed with basis on the EMCDG and on the parameters' purity information, in accordance to Algorithm 6.3; in short, the purified EMCDG is obtained by removing the edges representing safe and read-only parameters from the EMCDG.

Finally, the *purified Function Set* is defined with basis on the purified EMCDG, in accordance to the algorithm shown in Figure 6.4. Each entry in the Function Set table contains information on the types required (*child types* column) and types provided (*return types* column) by the corresponding member.

Figure 6.1 and Table 6.2 illustrate, respectively, the purified EMCDG and the purified Function Set generated for the `Stack` class; additionally, the original EMCDG and the entries excluded from the Function Set as a result of the Parameter Purity Analysis procedure are shown for comparison purposes. It should be noted that the purified Function Set for the `Stack` class includes only 7 entries, whereas the Function Set generated without using Purity Analysis contains 12 entries.

6.3 Experimental Studies

This Section describes the case studies implemented with the objectives of observing the impact of the Input Domain Reduction proposal – both in terms of the size of the input domain (Subsection 6.3.1) and of the results yielded by the *eCrash* tool (Subsection 6.3.2).

Algorithm 6.2: Algorithm for EMCDG generation with basis on the Test Cluster.

```
Data: test cluster
Result: extended method call dependence graph
begin insert vertices
    insert data type vertices;
    insert member vertices;
begin connect vertices
    begin insert directed edges from data type vertices to member vertices
        foreach data type vertex do
            foreach member vertex do
                if member is provider of data type then
                    insert insert edge from data type vertex to member vertex
                    labeled [RE, IP, Pn];
    begin insert directed edges from member vertices to data type vertices
        foreach member vertex do
            foreach data type vertex do
                foreach data type required by member do
                    insert insert edge from member vertex to data type vertex
                    labeled [IP, Pn];
    begin insert directed edges from data type vertices to data type vertices
        foreach array type vertex do
            foreach data type do
                if data type is array component then
                    insert insert edge from array vertex to data type vertex
                    labeled [AR];
        foreach interface vertex do
            search known implementing classes in test cluster;
            foreach implementing class do
                insert edge from interface vertex to implementing class vertex
                labeled [IM];
        foreach class vertex do
            search known subclasses in test cluster;
            foreach known subclass do
                insert insert edge from class vertex to subclass vertex
                labeled [EX];
    begin insert directed edges from constant vertices to data type vertices
        foreach data type vertex do
            foreach data type vertex do
                if constant is provider of data type then
                    insert insert edge from data type vertex to constant vertex
                    labeled [CO];
```

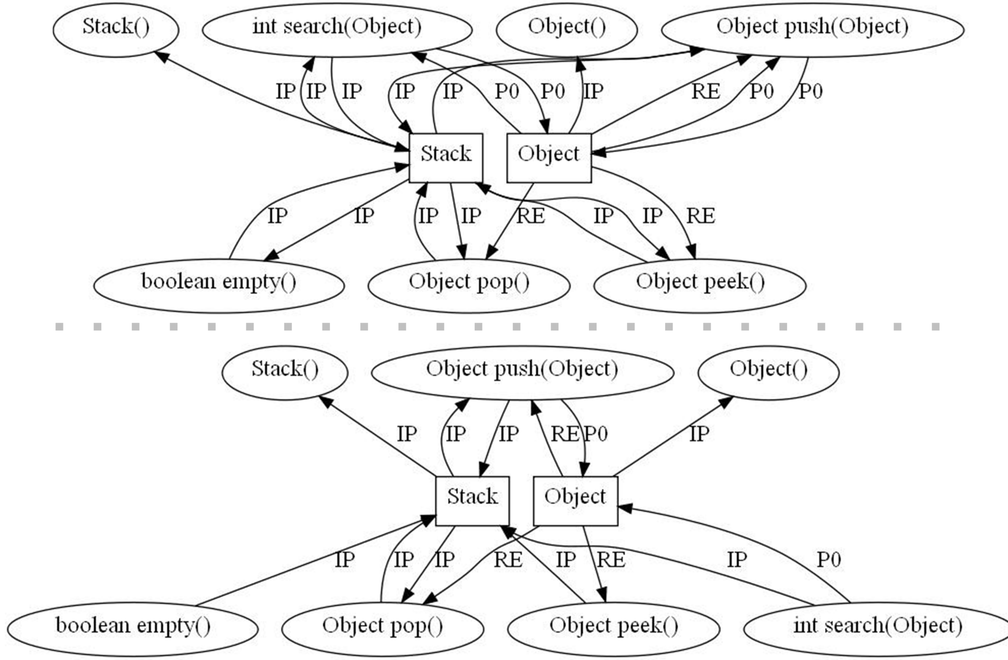

Figure 6.1: EMCDG *(top)* and purified EMCDG *(bottom)* for `Stack`.

Function Set

Member	*Return Type*	*Child Types*
Stack()	Stack [IP]	
Object pop()	Object [RE]	Stack [IP]
Object pop()	Stack [IP]	Stack [IP]
Object push(Object)	Object [RE]	Stack [IP], Object [P0]
Object push(Object)	Stack [IP]	Stack [IP], Object [P0]
Object peek()	Object [RE]	Stack [IP]
Object()	Object [RE]	
	Entries Excluded	
Object push(Object)	Object [P0]	Stack [IP], Object [P0]
Object peek()	Object [IP]	Stack [IP]
boolean empty()	Stack [IP]	Stack [IP]
int search(Object)	Stack [IP]	Stack [IP], Object [P0]
int search(Object)	Stack [P0]	Stack [IP], Object [P0]

Figure 6.2: Purified Function Set for the `Stack` class *(top)* and entries excluded from the Purified Function Set *(bottom)*.

Algorithm 6.3: Algorithm for the generation of the purified EMCDG.

```
Data: EMCDG, parameter purity information
Result: purified EMCDG
foreach EMCDG member node do
    foreach incoming edge do
        if Associated Item is not RETURN then
            parameterPurity ← get parameter purity;
            if parameterPurity is SAFE or READ-ONLY then
                remove incoming edge;
```

Algorithm 6.4: Algorithm for the generation of the purified Function Set with basis on the purified EMCDG.

```
Data: purified EMCDG
Result: purified function set
foreach EMCDG member node do
    create new function set entry for member;
    foreach outgoing edge do
        dataType ← get destination node;
        add dataType to member child types;
    foreach incoming edge do
        dataType ← get origin node;
        add dataType to member return types;
```

6.3.1 Input Domain Size Study

With our approach, the EMCDG models call dependencies, and the Function Set encompasses the entries from which the Test Data generation algorithm can choose when evolving Test Programs; as such, the impact of the Input Domain Reduction strategy proposed on the size of the search space is best assessed by comparing the purified Function Sets and EMCDGs to those obtained when no Parameter Purity Analysis is employed.

The statistics depicted in Table 6.5 for the `Stack` and `BitSet` classes show a clear reduction in the size of the input domain; the number of Function Set entries in the Purity column is only 65.2% of those obtained when no Parameter Purity Analysis is used.

This means that if Parameter Purity Analysis is not performed, and when searching for Test Cases for these classes, approximately a third of the set of instructions that can be selected for integrating the generated Test Programs have no (positive) impact on the definition of test scenarios.

	No Purity		Purity	
	EMCDG Edges	*Function Set Entries*	*EMCDG Edges*	*Function Set Entries*
`Stack`	19	12	14	7
`BitSet`	106	54	88	36

Table 6.5: Experimental results of the Input Domain Size case study: number of EMCDG edges and Function Set entries, obtained for the `Stack` and `BitSet` classes, with and without Parameter Purity Analysis.

6.3.2 Test Data Generation Results Study

In this Subsection, the results yielded by the *eCrash* tool when the Parameter Purity Analysis phase is included and excluded from the process are analysed and compared.

A single population of 10 individuals was used. 20 runs were executed for each of the MUTs. The MUTs' CFG nodes were initialized with a weight of 200, with the α and β parameters (Section 5.1) being set to 0.5 and 150, respectively. The search stopped if an ideal individual was found or after 100 generations. For breeding individuals, 3 pipelines were used: a Reproduction pipeline, a Crossover pipeline, and a Mutation pipeline; the probabilities of choosing these pipelines were set equal values. The selection method employed was Tournament Selection with a size of 2.

Table 6.6 presents the results obtained for the `Stack` and `BitSet` classes. For the `Stack` class, the number of generations required to attain full coverage using Parameter Purity Analysis was, on average, 2.6 – less than half of those required when no Parameter Purity Analysis was employed. All the runs yielded full coverage in both cases. For the `BitSet` class – and although 33.3% of the Function Set entries were eliminated when Parameter Purity Analysis was used – the improvement was not as clear. Still, the average percentage of Test Cases that accomplished full coverage within a maximum of 100 generations increased approximately 7%.

The graphs shown in Figure 6.3 represent the average number of CFG nodes left to be covered per generation. Again, the results obtained for the `Stack` yield a significant improvement, whereas those presented for the `BitSet` Test Object show a slight (but clear) improvement.

	No Purity		**Purity**	
	gens	*full*	*gens*	*full*
`Stack`				
pop	4.5	100%	1.3	100%
push	1.9	100%	1.9	100%
empty	7.1	100%	1.4	100%
peek	4.2	100%	1.3	100%
search	9.4	100%	6.9	100%
`BitSet`				
hashCode	1.6	100%	1.3	100%
clear(int)	43.3	55%	37.7	60%
clear()	15.8	100%	17.7	100%
clear(int,int)	-	0%	63.0	10%
toString	21.6	100%	18.4	100%
isEmpty	13.4	90%	4.2	90%
length	48.0	50%	47.4	95%
get(int,int)	75.5	15%	-	0%
get(int)	13.2	50%	41.3	60%
size	1.6	100%	1.3	100%
set(int,boolean)	13.5	100%	9.2	90%
set(int,int)	42.2	90%	36.6	90%
set(int,int,boolean)	42.0	90%	39.4	100%
set(int)	26.0	70%	25.4	90%
flip(int,int)	48.8	40%	57.7	35%
flip(int)	41.0	60%	33.5	60%
andNot	38.2	45%	26.0	70%
cardinality	10.9	100%	9.7	100%
intersects	58.5	40%	61.8	40%
nextSetBit	68.0	10%	56.0	45%
xor	34.9	70%	29.8	90%
Averages				
`Stack`	*5.4*	*100.0%*	*2.6*	*100.0%*
`BitSet`	*32.9*	*65.5%*	*30.9*	*72.6%*

Table 6.6: Experimental results for the Test Data Generation study: average number of generations required to attain full structural coverage (*gens*); and percentage of runs attaining full structural coverage (*full*); for the public methods of the `Stack` and `BitSet` classes, with and without Parameter Purity Analysis.

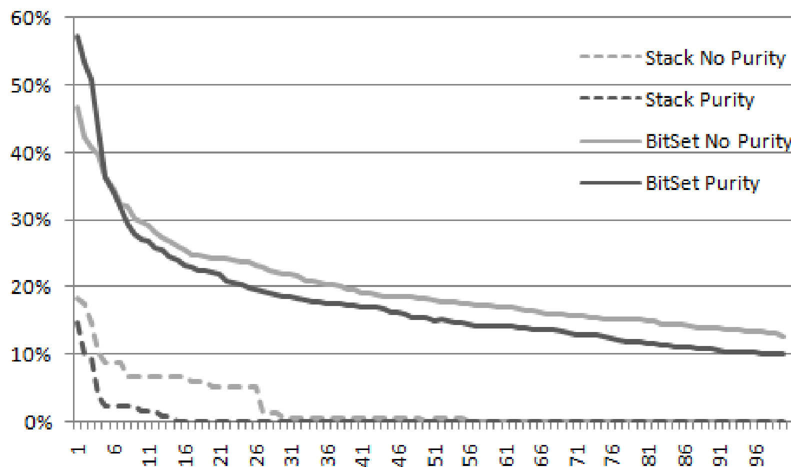

Figure 6.3: Experimental results for the Input Domain Reduction study: average percentage of CFG nodes remaining per generation, for the `Stack` and `BitSet` classes, with and without Parameter Purity Analysis.

6.3.3 Discussion

The results observed in the Input Domain Size experiment indicate that the search space of Evolutionary Testing problems can be dramatically reduced by embedding Parameter Purity Analysis into the process. For the Test Objects used, approximately a third of the set of entries that could be selected for integrating the generated Test Cases were discarded; these instructions would have no (positive) impact on the definition of test scenarios.

In terms of the results yielded by the *eCrash* tool when Parameter Purity Analysis is used, a significant improvement is clearly observable in terms of the efficiency of the search – i.e., fewer generations (and, consequently, less computational time) are required to find an adequate Test Set if the conditions are, otherwise, similar.

Finally, it should be mentioned that the Input Domain Reduction strategy proposed also enhances the Test Data Generation process indirectly, by preventing irrelevant instructions from obstructing the search by throwing runtime exceptions and rendering Test Cases unfeasible.

This fact is especially pertinent given that our Test Case Evaluation methodology does consider unfeasible Test Cases for breeding at certain points of the evolutionary search (cf. Chapter 5). The inclusion of a Parameter Purity Analysis phase into the process thus strengthens our Test Program Evaluation proposal, by ensuring that unfeasible MCSs are composed solely by instructions that are relevant in terms of state scenario definition.

6.4 Summary

An Input Domain Reduction methodology, based on the concept of Parameter Purity Analysis, for eliminating irrelevant variables from Object-Oriented Test Data generation search problems, was proposed. Our approach to Object-Oriented Evolutionary Testing involves representing Test Programs using the STGP paradigm; Purity Analysis is particularly useful in this context, as it provides a means to automatically identify and remove Function Set entries that do not contribute to the definition of interesting test scenarios; nevertheless, the concepts presented are generic and may be employed to enhance other search-based Test Data generation methodologies in a systematic and straight-forward manner.

The observations made indicate that the Input Domain Reduction strategy presented has a highly positive effect on the efficiency of the Test Case generation algorithm; less computational time is spent to achieve results. The process of “trimming” the input domain in order to eliminate irrelevant entries also ensures that Test Cases are not rendered unfeasible by the inclusion of unsuitable instructions; this strategy is thus of special importance, given that our Test Data Evaluation strategy does consider unfeasible Test Cases at certain stages of the search.

Chapter 7

An Adaptive Approach to the Evolutionary Testing of Object-Oriented Software

Evolutionary Algorithms are powerful – yet general – methods for search and optimization [BFM97]. Their generality comes from the unbiased nature of the standard operators used, which perform well for problems where little or no domain knowledge is available [Ang95]. However, if knowledge about a problem is available, a bias can be introduced directly into the problem so as to remove (or penalise) undesirable candidate solutions and improve the efficiency of the search.

Unfortunately, *a priori* knowledge about the intricacies of the problem is frequently unavailable. Having little information about a problem does not, however, necessarily prevent introducing an appropriate specific bias into an evolutionary problem; for many tasks, it is possible to dynamically adapt aspects to anticipate the regularities of the environment and improve solution optimization or acquisition speed. Adaptive Evolutionary Algorithms are distinguished by their dynamic manipulation of selected parameters or operators during the course of evolving a problem solution [HME97]. They have an advantage over their standard counterparts in that they are more reactive to the unanticipated particulars of the problem and, in some formulations, can dynamically acquire information about regularities in the problem and exploit them.

Typically, Evolutionary Algorithms maintain a population of candidate

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. Mutation is the main process through which new genetic material is introduced during an evolutionary run with the intent of diversifying the search and escaping local maxima. The main contribution of this Chapter is that of proposing an adaptive strategy for promoting the introduction of relevant instructions into the existing Test Programs by means of Mutation; the set of instructions from which the algorithm can choose is ranked, with their rankings being updated every generation in accordance to the feedback obtained from the individuals evaluated in the preceding generation.

This Chapter is organised as follows. The next Section provides background on Adaptive Evolutionary Algorithms; in Section 7.2, the Adaptive Evolutionary Testing strategy proposed is presented and detailed; the experiments performed in order to validate the technique are discussed in Section 7.3; and Section 7.4 summarises the main contributions of this study.

7.1 Adaptive Evolutionary Algorithms

The action of determining the variables and parameters of an Evolutionary Algorithm to suit the problem has been termed *adapting* the algorithm to the problem; in Evolutionary Algorithms this can be performed dynamically, while the algorithm is searching for a solution.

Adaptive Evolutionary Algorithms provide the opportunity to customise the Evolutionary Algorithm to the problem, and to modify the configuration and the strategy parameters used while the problem solution is sought. This enables incorporating domain information into the Evolutionary Algorithm more easily, and allows the algorithm itself to select those parametrisations which yield better results; also, these values can be modified during the run to suit the situation.

Adaptive Evolutionary Algorithms have already been applied to solve several search problems; interesting review articles include [Ang95, HME97]. In [HME97], Hinterding *et al.* proposed a classification based on the adaptation type and adaptation level of the Adaptive Evolutionary Algorithm. The type of adaptation consists of two main categories: static and dynamic. *Static adaptation* is where the strategy parameters have a constant value throughout the run of the Evolutionary Algorithm; consequently, an external agent or mechanism (e.g., the user) is needed to tune the desired

strategy parameters and choose the most appropriate values. *Dynamic adaptation* happens if there is some mechanism which modifies a strategy parameter without external control, and can be divided further into deterministic, adaptive, and self-adaptive mechanisms.

Deterministic dynamic adaptation is employed if the value of a strategy parameter is altered by some deterministic rule, without using any feedback (e.g., using a time-varying schedule). *Adaptive dynamic adaptation* takes place if there is some form of feedback from the Evolutionary Algorithm that is used to determine the direction and/or magnitude of the change to the strategy parameter (e.g, involving credit assignment). With *dynamic self-adaptation*, the idea of the "evolution of evolution" is used; the parameters to be adapted are encoded onto the individual, and undergo Mutation and Recombination themselves (e.g., Meta-GP [Edm01]).

The level of adaptation consists of the following categories. *Environment level adaptation* happens when the response of the environment to the individual is changed (e.g, when the penalties in the fitness function change). In *population level adaptation*, some or all of the global parameters are modified – i.e., those that apply to all members of the population (e.g., global Mutation and Crossover frequency). *Individual level adaptation* adjusts strategy parameters held within individuals and whose value affects only that individual (e.g., the Crossover point). *Component-level adaptation* adjusts strategy parameters local to some component or gene of an individual in the population.

Several methodologies to the Evolutionary Testing of Object-Oriented software have been proposed, focusing on the usage of distinct Evolutionary Algorithms (cf. Chapter 3). However, to the best of the author's knowledge, there are no studies on the possibility of applying Adaptive Evolutionary Algorithm to Evolutionary Testing problems; in the following Sections, a novel population-level adaptive dynamic adaptation technique will be detailed.

7.2 Adaptive Evolutionary Testing Strategy

STGP-based approaches to Object-Oriented Evolutionary Testing involve encoding candidate solutions as STGP trees; each tree subscribes to a Function Set, which must be specified beforehand and establishes the constraints involved in the trees' construction. In other words, the Function Set contains the set of instructions from which the algorithm can choose when building Test Programs.

The Function Set can be defined completely automatically based solely on the Test Cluster information (cf. Section 4.3.3). The definition of the Test Cluster is, therefore, of paramount importance to the algorithm's performance and accuracy; however, if the Test Cluster consists of many classes (or if it is composed of few classes which possess a high number of public methods), the Function Set can be extremely large.

With an increasing size of the Function Set (and hence an increasing size of the search space) the probability that the "right" methods appear in a candidate test sequence decreases – and so does the efficiency of the evolutionary search. Conversely, if a more conservative strategy is employed, the Test Cluster may not include all the classes needed to attain full coverage, thus compromising effectiveness. As such, the selection of the classes and methods to be included in the Test Cluster – and, consequently, in the Function Set – must be carefully pondered, and adequate strategies must be employed for defining the Test Cluster and sampling the search domain.

Still, there are good reasons to suppose that there is no one strategy, however clever, recursive, or self-organising that will be optimal for all problem domains; the Test Cluster parametrisation process is heavily problem-specific and, as such, it usually depends on the users' decisions. Leaving this task to the user has, however, several drawbacks. Namely [HME97]:

- the users' mistakes in setting the parameters could be sources of errors and/or suboptimal performance;
- parameter tuning costs a lot of time; and
- the optimal parameter value may vary during the evolution.

What's more, the users' choices are inevitably biased, and performance is (arguably) often compromised for the sake of accuracy; in the particular case of Evolutionary Testing problems, not doing so could result in the impossibility of obtaining suitable Test Sets, in conformity to the criteria defined.

In [Wap07], Wappler suggested the following strategies for addressing the problem of large Function Sets, that result from large Test Clusters with classes that possess many methods:

- Performing a static analysis so as to eliminate all the functions in the Function Set that correspond to methods which are neither object-creating nor state-changing. An Input Domain Reduction strategy,

based on the concept of Purity Analysis, that meets this suggestion has already been proposed in Chapter 6.

- Defining a distance-based heuristic, that prevents the methods from those Test Cluster classes that are associated to the CUT via several other classes from being transformed to functions of the Function Set. Such an heuristic would have to be problem-specific, and decisions would have to be made statically and *a priori* – potentially compromising the success of the search. It seems difficult to implement an automated solution for this idea without compromising generality.

- Naming classes whose methods shall not be transformed to functions of the Function Set. This idea exploits the user's knowledge of the CUT, and suffers from the drawbacks mentioned above.

We propose a distinct strategy, based on the concept of dynamically adapting the Function Set's constraints selection probabilities. During an evolutionary run, it is possible to perceive that the introduction of certain instructions should be favoured. By allowing the constraints' selection probabilities to variate throughout the search, with basis on the feedback obtained by the behaviour of the individuals produced and evaluated previously, the introduction of interesting genetic material will be promoted. This strategy presents the following advantages:

- it permits mitigating the negative effects of including a large number of entries into the Test Cluster; and

- it allows a higher degree of freedom when defining the Test Cluster, by minimizing the impact of redundant, irrelevant or erroneous choices – especially those made by the user.

Mutation plays a central role on the diversification of the search and on the exploration of the search space; it basically consists of selecting a Mutation point in a tree, and substituting the sub-tree rooted at the point selected with a newly generated sub-tree [Koz92]. Previous studies indicate that better results can be attained if the Mutation operator is assigned a relatively high probability of selection (cf. Chapter 5).

Mutation is, in fact, the main process by which new genetic material is introduced during the evolutionary search. In the particular case of Object-Oriented Evolutionary Testing problems, it allows the introduction of new sequences of method calls into the generated Test Programs, so as to allow

trying out different objects and states in the search for full structural coverage. Nevertheless, it is clear that during an evolutionary run, it is possible to perceive that some method calls are more relevant than others, e.g. because they had been less prone to throw runtime exceptions and their introduction will likely contribute to Test Case feasibility, or simply because they have been used less frequently and their introduction will promote diversity (precisely the main task of the Mutation operator).

Whenever Mutation occurs, a new (sub-)tree must be created; usually, one of the standard tree builders (e.g., Grow, Full, Half-Builder or Uniform) is used to generate these trees [Luk00a]. We propose employing Luke's Probabilistic Tree Creation 2 (PTC2) algorithm [Luk00b] to perform this task, so as to take advantage of the built-in feature that allows assigning probabilities to the selection of constraints. Also, and more importantly, we have modified this algorithm in order to be able to dynamically update the constraints' probabilities during the evolutionary run.

PTC2 provides uniform distribution of functions and has very low computational complexity [Luk00a]. Also – and most interestingly – PTC2 has provisions for picking non-terminals with a certain probability over other non-terminals of the same return type, and terminals over other terminals likewise. In order to illustrate the methodology followed by this algorithm, let us consider a simple problem which includes a Function Set (Table 7.1) composed of seven entries (or constraints), defining three non-terminal nodes – `void print(String)`, `String intToStr(Integer)`, `Integer add (Integer, Integer)` – and four terminal nodes – `"Foo"`, `"Bar"`, `0` and `1`. Also, it defines three atomic types – TREE, STRING and INT – and one set type – OBJECT, which includes both INT and STRING. The TREE type is used as return type of the STGP tree.

The constraint selection rankings are also defined. `"Foo"` is given a rank of 0.8, and `"Bar"` a rank of 0.2, for example; this means that, if the PTC2 algorithm is required to select a terminal node with a STRING return type, it will select constraint `"Foo"` with a probability of 80% and `"Bar"` with a probability of 20%. If, however, it is required to select a terminal node with an OBJECT return type, PTC2 uniformly distributes the rankings of the STRING and INT atomic types, with the constraints probabilities being defined as follows: `"Foo"`–40%; `"Bar"`–10%; `0`–20%; `1`–30%.

Continuing with this example, if required to grow a tree of size 3, the PTC2 algorithm would build the tree depicted in Figure 7.1 with a 19.2% chance: 100% probability of selecting the root node, times 80% probability of selecting the non-terminal constraint `Integer add(Integer, Integer)`

Function Set and Type Set

Function Name	*Return Type*	*Child Types*
void print(Object)	TREE	OBJECT
String intToStr(Integer)	STRING (rank:0.2)	INT
“Foo”	STRING (rank:0.8)	
“Bar”	STRING (rank:0.2)	
Integer add(Integer,Integer)	INT (rank:0.8)	INT, INT
0	INT (rank:0.4)	
1	INT (rank:0.6)	
Set Types: OBJECT = [STRING, INT]		

Table 7.1: Example Function Set and Type Set.

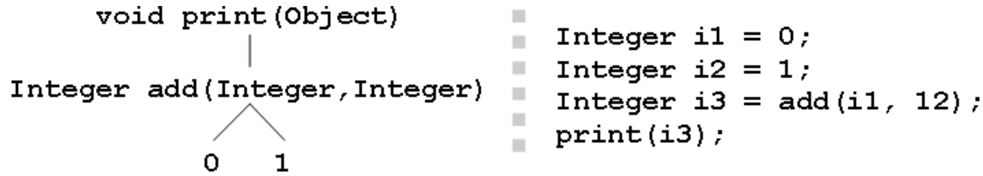

Figure 7.1: Example MCT *(left)* and corresponding Test Program *(right)*, built using the Function Set defined in Table 7.1.

as an OBJECT type provider for the root node, times 40% chance of choosing `0` as the first terminal of type INT, times 60% chance of selecting `1` as the second terminal.

The dynamic adaptive strategy described in the following Subsection aims at dynamically tuning the Function Set's constraints selection rankings, so as to promote the creation of sub-trees, for insertion in the population via Mutation, that favour both feasibility and diversity.

7.2.1 Constraint Selection Ranking Adaptation Strategy

Let the *constraint selection ranking* of constraint c in generation g be identified as ρ_c^g. Also, let λ be the *runtime exceptions caused factor*, σ be the *runtime exceptions caused by ancestors factor*, and γ be the *constraint diversity factor*. Then, ρ_c^g is updated, at the beginning of each generation, in accordance to the following Equation.

$$\rho_c^g = \rho_c^{g-1} - \lambda_c^{g-1} - \sigma_c^{g-1} - \gamma_c^{g-1} \tag{7.1}$$

That is, the constraint selection ranking ρ_c^g of a given constraint c in generation g is calculated as being the constraint selection ranking ρ of the

previous generation, minus the λ factor of the previous generation (with $\lambda \in [0, 1]$), minus the σ factor of the previous generation (with $\sigma \in [0, 1]$), minus the γ factor of the previous generation (with $\gamma \in [-1, 1]$).

In order to calculate the normalised constraint selection ranking ρ'^{g}_{c}, if the minimum ρ^{g}_{c} in generation g is negative, the data is firstly shifted by adding all numbers with the absolute of the minimum ρ^{g}_{c}; then, ρ'^{g}_{c} is normalised into the range of $[0, 1]$ as follows.

$$n'^{g}_{c} = \frac{n^{g}_{c}}{n^{g}_{MAX} - n^{g}_{MIN}} \tag{7.2}$$

The following subsections detail the procedure used for calculating the λ, σ, and γ factors.

Runtime Exceptions Caused Factor Let E^{g}_{c} be the set of runtime exceptions caused by constraint c in generation g, and T^{g} be the set of trees produced in generation g, with $|E^{g}_{c}|$ and $|T^{g}|$ being their cardinalities. Then, λ is calculated as follows.

$$\lambda^{g}_{c} = \frac{|E^{g}_{c}|}{|T^{g}|} \tag{7.3}$$

That is, the λ factor is equal to the number of runtime exceptions thrown by instructions corresponding to constraint c, dividing by the total number of trees. It should be noted that only a single runtime exception may be thrown by a Test Program (i.e., by a MCT).

This factor's main purpose is that of penalising the ranking of constraints corresponding to instructions that have caused runtime exceptions to be thrown in the preceding generation. This factor is normalised into the range of $[0, 1]$ using Equation 7.2.

Runtime Exceptions Caused by Ancestors Factor Let X^{g}_{c} be the set of runtime exceptions thrown by ancestors of constraint c in generation g, and $x^{g}_{c^a} \in X^{g}_{c}$ be a runtime exception thrown by an ancestor of level a, with $a \in \{2 = parent, 3 = grandparent, \ldots\}$ being the *ancestry level* of the constraint that threw the exception. Also, let A^{g}_{c} be the multiset containing the ancestry levels of $x^{g}_{c^a} \in X^{g}_{c}$. Then, σ is calculated as follows.

$$\sigma^{g}_{c} = \sum_{a \in A^{g}_{c}} a^{-1} \tag{7.4}$$

That is, the σ factor is equal to the sum of the inverses of the ancestry levels of the ancestors of constraint c that threw runtime exceptions.

This factor's main purpose is that of penalising the ranking of constraints corresponding to instructions which have participated in the composition of sub-trees (i.e., sub-MCSs) that have caused runtime exceptions to be thrown in the preceding generation; the higher the ancestry level, the lower the penalty. This factor is normalised into the range of $[0, 1]$ using Equation 7.2.

Constraint Diversity Factor Let C^g be a multiset containing the number of times each constraint appeared in generation g, and c^g be the number of times constraint c appeared in generation g. Also, let m_{C^g} be the mean of the values contained in multiset C^g, and $d_c^g = c^g - m_{C^g}$ be the deviation of constraint c in generation g, and $r_d^g = d_{MAX}^g - d_{MIN}^g$ be the range of deviation for generation g. Then, γ_c^g is calculated as follows.

$$\gamma_c^g = \frac{d_c^g}{r_d^g} \tag{7.5}$$

That is, the γ factor is equal to the absolute deviation between constraint c's number of appearances and the mean number of all constraints appearances, dividing by the range of deviation for generation g.

This factor's main purposes are those of allowing constraints to recover their ranking if they have been used infrequently, and penalising the ranking of constraints which have been selected too often.

7.3 Experimental Studies

The adaptive strategy described in the preceeding Section was embedded into the *eCrash* automated Evolutionary Testing tool, with the objective of observing the impact of this technique on both the efficiency and effectiveness of the Test Data generation process.

The Java `Vector` and `BitSet` classes (JDK 1.4) were used as Test Objects. The experiments were executed using an Intel Core2 Quad 2.60GHz processor with 4.0 GB RAM desktop, with 4 Test Data generation processes running in parallel. 20 runs were executed for each of the 67 MUTs – in a total of 820 runs for the `Vector` class and 520 runs for the `BitSet` Class. Half of these runs were executed employing the adaptive strategy proposed, and half using a "static" approach for comparison purposes. The only difference between the adaptive and the static runs was that, in the latter, the constraints' rankings remained unaltered throughout the evolutionary search. Since the same seeds were used in both the adaptive and non-adaptive runs,

and because *eCrash* is deterministic, the discrepancies in the results will solely mirror the impact of the adaptive technique employed.

A single population of 10 individuals was used; the rationale for selecting a relatively small population size had to do with the adaptive algorithm's need of obtaining frequent feedback. The search stopped if an ideal individual was found or after 200 generations. For the generation of individuals, 3 child sources were defined: strongly-typed versions of Mutation (selection probability: 40%) and Crossover (selection probability: 30%), and a simple Reproduction operator (selection probability: 30%). The selection method was Tournament Selection with size 2. The tree builder algorithm was PTC2 (for the reasons explained in the preceding Section), with the maximum and minimum tree depths being defined as 1 and 4. The constraints' ranking were initialized with the value 1.0, and were updated at the beginning of every generation, before individuals were produced (cf. Algorithm 4.1 on page 54), in accordance to Equation 7.1.

Table 7.2 depicts the percentage of successful runs (i.e., runs in which a Test Set attaining full structural coverage was found) for the MUTs of the `Vector` and `BitSet` classes, with and without adaptation. The graphs shown in Figure 7.2 contain the percentage of CFG nodes remaining per generation using the adaptive and the static techniques for the classes tested; their inclusion enables the analysis of the strategy's impact during the course of the search.

The results depicted in Table 7.2 clearly indicate that the test case generation process's performance is improved by the inclusion of the Adaptive Evolutionary Testing methodology proposed. The adaptive strategy outperformed the static approach for 28.4% of the MUTs tested, whereas the latter only surpassed the former in 5.9% of the situations. In terms of the average success rate, the adaptive strategy enhances results by 3% for the `Vector` class; the improvement is even more significant for the `BitSet` class, with the results meliorating 11%.

What's more, the adaptive strategy allowed attaining full structural coverage in some situations in which the success rate had been of 0% using the non-adaptive strategy – namely, for the `Object remove(int)` and `List subList(int,int)` MUTs of the `Vector` class, and for the `int length()` and `boolean intersects(BitSet)` MUTs of the `BitSet` class; these observations indicate that this strategy is specially suited for overcoming some difficult state problems.

The graph shown in Figure 7.2 also provides clear indication that the evolutionary search benefits from the inclusion of the adaptive approach

MUT	*adaptive*	*static*
`Vector`		
void add(int,Object)	80%	**90%**
boolean add(Object)	100%	100%
Object get(int)	100%	100%
int hashCode()	100%	100%
Object clone()	0%	0%
int indexOf(Object)	100%	100%
int indexOf(Object,int)	**20%**	10%
void clear()	100%	100%
boolean equals(Object)	100%	100%
String toString()	100%	100%
boolean contains(Object)	**50%**	40%
boolean isEmpty()	100%	100%
int lastIndexOf(Object,int)	0%	0%
int lastIndexOf(Object)	100%	100%
boolean addAll(Collection)	**90%**	70%
boolean addAll(int,Collection)	**30%**	20%
int size()	100%	100%
Object[] toArray()	100%	100%
Object[] toArray(Object[])	40%	40%
void addElement(Object)	100%	100%
Object elementAt(int)	100%	100%
Object remove(int)	**20%**	0%
boolean remove(Object)	100%	100%
Enumeration elements()	100%	100%
Object set(int,Object)	**100%**	80%
int capacity()	100%	100%
boolean containsAll(Collection)	100%	100%
void copyInto(Object[])	100%	100%
void ensureCapacity(int)	100%	100%
Object firstElement()	100%	100%
void insertElementAt(Object,int)	80%	**90%**
Object lastElement()	100%	100%
boolean removeAll(Collection)	100%	100%
void removeAllElements()	100%	100%
boolean removeElement(Object)	30%	**40%**

Continued on next page

Table 7.2 – continued from previous page

MUT	*adaptive*	*static*
void removeElementAt(int)	**20%**	10%
boolean retainAll(Collection)	100%	100%
void setElementAt(Object,int)	**100%**	70%
void setSize(int)	100%	100%
List subList(int,int)	**30%**	0%
void trimToSize()	100%	100%
`BitSet`		
boolean get(int)	**90%**	60%
BitSet get(int,int)	0%	0%
int hashCode()	100%	100%
Object clone()	0%	0%
void clear(int, int)	0%	0%
void clear()	100%	100%
void clear(int)	**90%**	80%
boolean equals(Object)	0%	0%
String toString()	100%	100%
boolean isEmpty()	100%	100%
int length()	**30%**	0%
int size()	100%	100%
void set(int)	**100%**	70%
void set(int, boolean)	100%	100%
void set(int, int)	**70%**	40%
void set(int, int, boolean)	40%	**70%**
void flip(int, int)	**60%**	20%
void flip(int)	**90%**	50%
void and(BitSet)	0%	0%
void andNot(BitSet)	**60%**	30%
int cardinality()	100%	100%
boolean intersects(BitSet)	**20%**	0%
int nextClearBit(int)	0%	0%
int nextSetBit(int)	10%	10%
void or(BitSet)	0%	0%
void xor(BitSet)	**90%**	30%

Table 7.2: Experimental Results for the Adaptive Evolutionary Testing study: percentage of runs attaining full structural coverage; for the public methods of the `Vector` and `BitSet` classes; with and without adaptation.

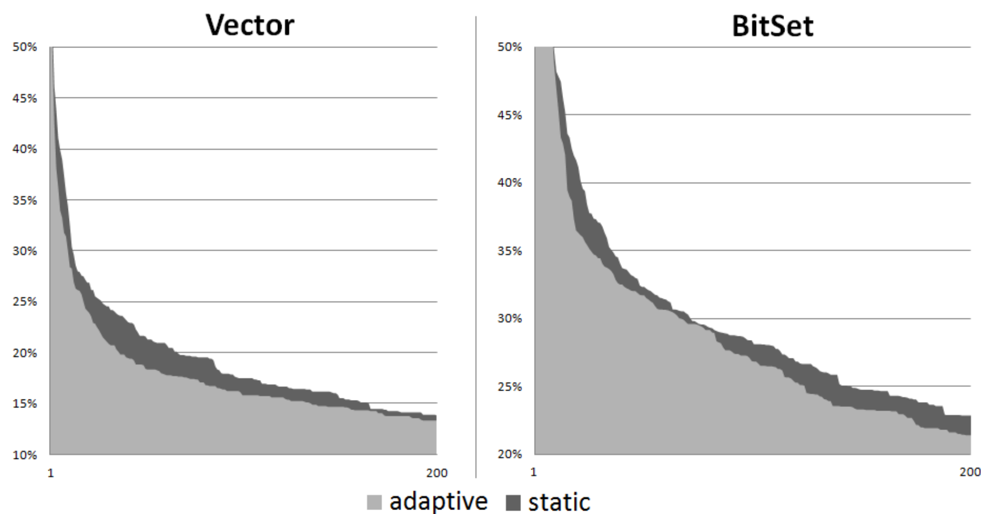

Figure 7.2: Average percentage of CFG nodes remaining per generation; for the public methods of the `Vector` and `BitSet` classes; with and without adaptation.

described. For the `Vector`'s MUTs, the average number of nodes remaining when the Adaptive Evolutionary Testing approach is used decreases as much as 6% during the initial generations; for the `BitSet` class, the contrast is the results is less perceptible, but the adaptive approach still manages to attain a 3% improvement at certain stages.

In terms of speed, the overhead introduced by embedding the adaptive strategy into the evolutionary algorithm was negligible; each generation took, on average, 23.25 seconds using the adaptive methodology, and 23.21 seconds using the static approach. The time overhead introduced by the adaptation procedure was a mere 0.19%.

7.4 Summary

Recent research on Evolutionary Testing has relied heavily on GP for representing and evolving Test Data for Object-Oriented software. The main contribution of this work is that of proposing a dynamic adaptation strategy for promoting the introduction of relevant instructions into the generated Test Programs by means of Mutation.

The Adaptive Evolutionary Testing strategy proposed obtains feedback from the individuals produced and evaluated in the preceding generation, and dynamically updates the selection probability of the constraints defined in the Function Set so as to encourage the selection of interesting

genetic material and promote diversity and Test Program feasibility. The experimental studies implemented indicate a considerable improvement in the algorithm's efficiency when compared to its static counterpart, while introducing a negligible overhead.

Chapter 8

Enabling Object Reuse on Genetic Programming-based Approaches to Object-Oriented Evolutionary Testing

The goal of Evolutionary Testing is to find a set of Test Programs that satisfies a certain test criterion. If structural adequacy criteria are employed, the basic idea is to ensure that all the control elements in a program are executed by a given Test Set, providing evidence of its quality. Object Reuse (OR) is a feature of paramount importance in this context.

Object Reuse means that one instance can be passed to multiple methods as an argument, or multiple times to the same method as arguments [Wap07]. In the context of Object-Oriented Evolutionary Testing, it enables the generation of Test Cases that exercise specific structures of software that would not be reachable otherwise.

The `equals` method of Java's `Object` class [Sun03] provides a paradigmatic example. Class `Object` is the root of the Java class hierarchy, and the `equals` method is used to assess if two objects are equivalent; also, several search methods rely on it to verify if an item is present in a collection (e.g., `Vector`'s `indexOf`). However, the `equals` method implements the most discriminating possible equivalence relation on objects: for any non-`null` reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same reference. This means that, in order for the method

```
Object object1 = new Object();
Object object2 = new Object();
boolean isEqual = object1.equals(object2);
System.out.println(isEqual);
```

```
Object object1 = new Object();
boolean isEqual = object1.equals(object1);
System.out.println(isEqual);
```

Listing 8.1: Programs exemplifying object equality verification in Java; the output of the program at the top is "false", whereas the output of the program at the bottom is "true".

`equals` to return `true`, the same `Object` reference must be passed as an argument twice – in the place of both the implicit parameter (i.e., the `this` parameter) and the explicit parameters. The programs depicted in Listing 8.1 illustrate this characteristic.

Also, every class has `Object` as a superclass; this means that every class inherits the `equals` method, and uses it internally for equivalence verification. `Object` subclasses may override `equals` in order to implement a less stringent equivalence relation. Still, it is not mandatory; what's more, recent studies have concluded that implementations of the `equals` methods are often faulty [VTFD07].

Recent research on Evolutionary Testing has relied heavily on GP for representing and evolving Test Data (cf. Chapter 3). However, standard GP approaches do not allow node reuse; in this Chapter a novel methodology to overcome this limitation is proposed, which involves the definition of novel type of GP nodes – the *At-Nodes* – that "point to" other nodes, thus effectively enabling the creation of edges to nodes that are already part of the tree, and allowing the reuse of sub-trees. The introduction of At-Nodes is performed by means of a custom-made evolutionary operator – the *Object Reuse operator*. This operator acts on an individual by selecting two nodes – the node to be replaced by the At-Node, and the node to be "pointed at" by the At-Node – and by inserting the newly created At-Node into the tree. At-Nodes may be removed from a tree by means of the *Reverse Object Reuse operator* which, in short, searches the tree for At-Nodes, and replaces these nodes with copies of the sub-trees pointed at by the At-Nodes. This particular operator allows avoiding the reformulation of other common biology-inspired mechanisms (e.g., Mutation and Crossover [Koz92]).

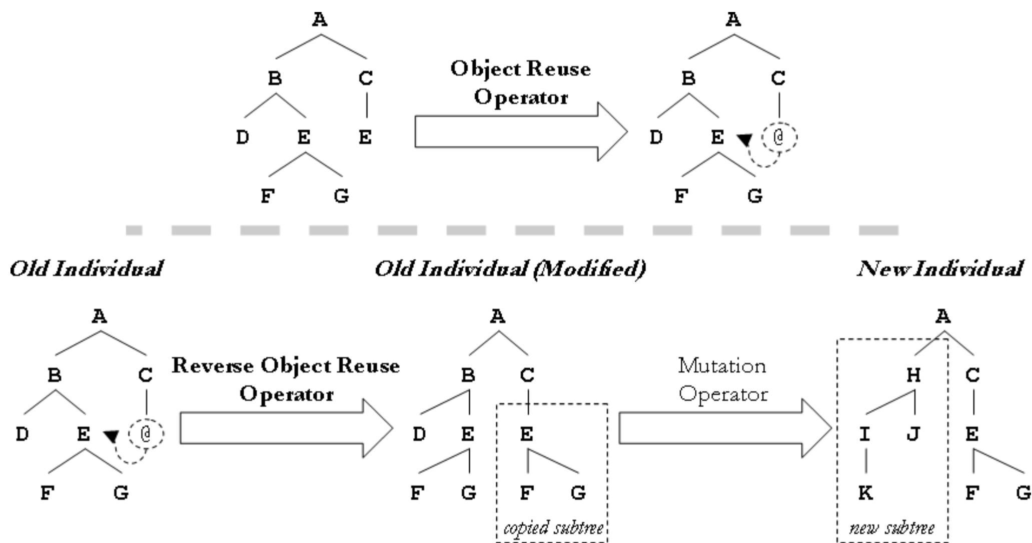

Figure 8.1: Object Reuse (*top*) and Reverse Object Reuse (*bottom*) operators overview.

In addition allowing specific structures to be traversed, the Object Reuse methodology proposed is able to enhance the performance of the Test Data generation process:

- it yields solutions with smaller overall size and lower average structural complexity;
- and the feasibility of the generated Test Programs is increased as a result of the introduction of a specific heuristic for node selection.

The Object Reuse methodology proposed is based on the introduction of two novel evolutionary operators: the Object Reuse Operator (detailed in the following Subsection), and the Reverse Object Reuse Operator (described in Subsection 8.2). Figure 8.1 provides an overview of these operators. In Section 8.3 the experiments performed in order to assess the impact of the reuse strategy proposed on the Test Data generation process are explained and discussed; Section 8.4 contextualises relevant related work, and the final Section summarises the contributions of this Chapter.

8.1 The Object Reuse Operator

Test Program quality evaluation on GP-based approaches to Object-Oriented Evolutionary Testing typically involves executing the generated Test Pro-

grams with the intention of collecting trace information with which to derive coverage metrics. Test Program execution requires decoding an individual's genotype (i.e., the MCT) into its phenotype (i.e., the Test Program). Figure 8.2 exemplifies this process; Object Reuse has not been introduced at this point. The MUT is the `indexOf` method of the `Vector` class – which corresponds to the root node of the MCT depicted in Figure 8.2a. Each node's parameters are provided by its children; the MCS (Figure 8.2b) corresponds to the linearised MCT, with tree linearisation being performed by means of a depth-first traversal algorithm (depicted in Algorithm 4.2 on page 68). Each MCS entry contains a *MIO*, which encloses: the method signature data necessary for the Test Program's source code to be assembled; and references to other MIOs providing the parameters (if any) for that method (enumerated between square brackets). The Test Program (Figure 8.2c) is computed with basis on the MCS and corresponds to a syntactically correct translation of the latter.

The purpose of the Object Reuse Operator is that of inserting At-Nodes into valid locations of a MCT; the concept of At-Node is, thus, key to the Object Reuse methodology proposed.

8.1.1 At-Nodes

At-Nodes are GP nodes that refer to other (standard) GP nodes, thus enabling the reuse of portions of the tree and, specifically, the reuse of the object references returned by the functions corresponding to the reused subtrees. This is accomplished by having the node pointed at by the At-Node provide the parameter not only to its parent node, but also to the At-Node's parent node; parameter assignment is performed during the MCT's linearisation by means of the process described in Subsection 8.1.4.

Figure 8.3a contains an example of a possible MCT resulting from the application of the Object Reuse operator to the tree depicted in Figure 8.2a. The At-Node labeled 0.1 replaces the node with the same label existing in the original MCT, whereas node 0.0.1 was selected as the node to be reused. As such, the `Object` instance returned by node 0.0.1 will be used both by its parent (labeled 0.0) and by the At-Node's parent (labeled 0). The MCS and Test Program shown in Figure 8.3b and 8.3c mirror this alteration: in the former, the MIO 0.0.1 provides the argument for the explicit parameters of both the 0.0 and 0 MIOs; and in the latter, the reference to the `Object` instance created at instruction 2 is passed to both the `add` and `indexOf` methods (instructions 3 and 4).

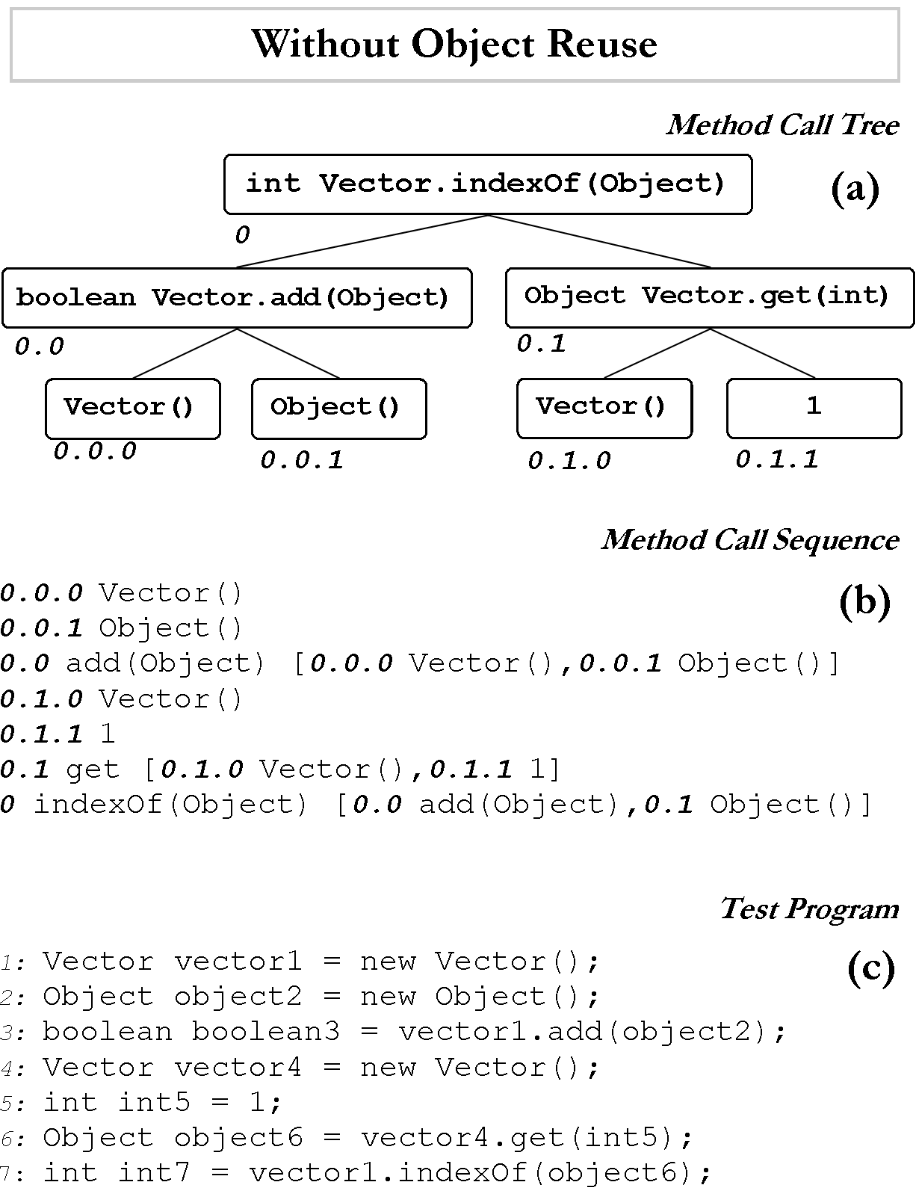

Figure 8.2: Example MCT without Object Reuse (*a*); and corresponding MCS (*b*) and Test Program (*c*).

The creation of an At-Node for posterior introduction into a MCT requires the Object Reuse operator to select two MCT nodes in the original tree: the *Destination Node* (i.e., the node to which At-Node points to) and the *Replaced Node* (i.e., the root node of the subtree to be truncated and substituted by the At-Node). The first task of the Object Reuse Operator is precisely that of indexing all the valid Replaced-Destination node pairs in a MCT.

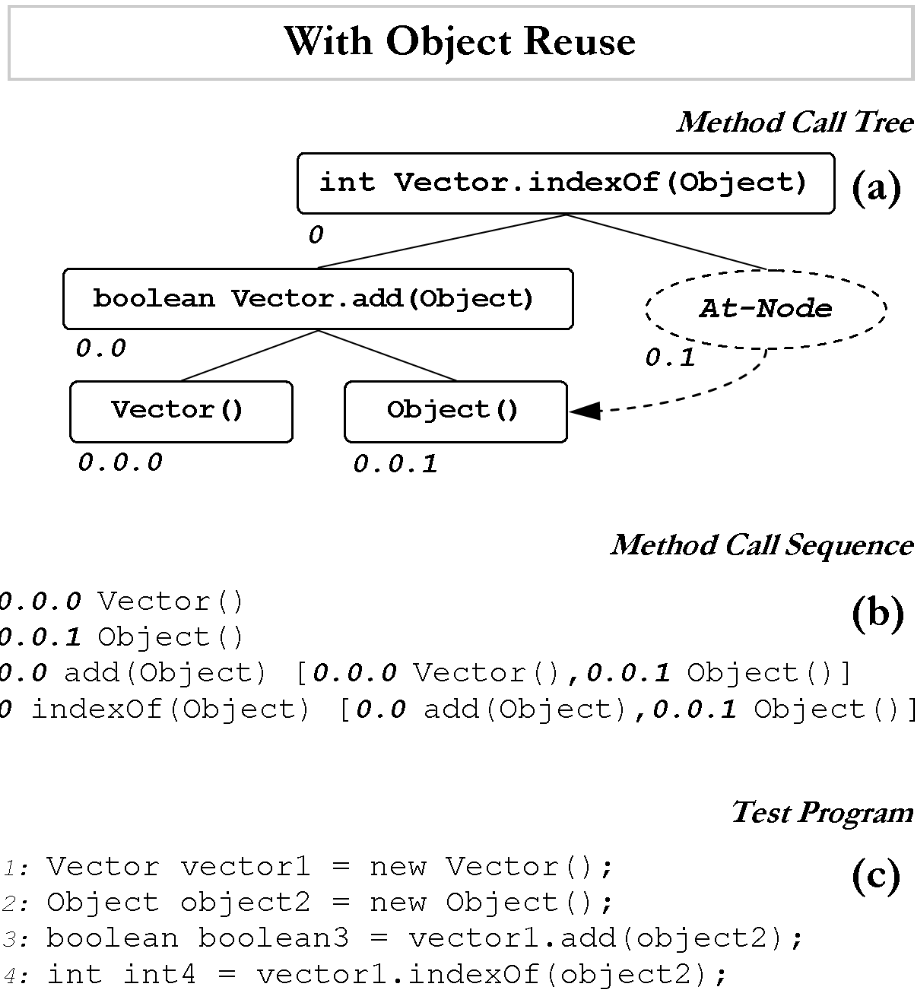

Figure 8.3: Example MCT with Object Reuse (a); and corresponding MCS (b) and Test Program (c).

8.1.2 Valid Replaced-Destination Node Pairs

A Replaced-Destination node pair is valid if:

- both nodes are distinct non-root standard GP nodes;
- the Replaced Node possesses a type that is swap-compatible with the Destination Node (e.g., a node of type `Vector` is swap-compatible with a node of type `Object`, because `Vector` is a sub-type of `Object`);
- the sub-tree rooted at the Replaced Node does not contain a node that is pointed at by an existing At-Node. When an At-Node is inserted into the tree, the sub-tree rooted at the Replaced Node is truncated;

if it contains a node that is already being reused, this operation will render the tree invalid;

- the Replaced Node is in a position reached by the linearisation algorithm prior to the Destination Node. This validation ensures that the MIOs only contain parameter references to elements that precede them in the MCS, and that the corresponding Test Program's method calls have their parameters provided by previously created instances.

After all the valid Replaced-Destination node pairs have been indexed, the Object-Reuse Operator proceeds to select one of those pairs.

8.1.3 Replaced-Destination Node Pair Selection

The node pair selection procedure is performed differently according to the individual's feasibility:

- if the individual is *feasible*, a Replaced-Destination node pair is chosen at random from the set of valid Replaced-Destination node pairs;
- if the individual is *unfeasible*, the Object Reuse operator attempts to select a valid pair so that the Replaced Node belongs to the non-executed portion of the tree, and the Destination Node belongs to the executed portion of the tree. If such pairs exist, one is selected at random; otherwise, a node pair is chosen at random from the set of all valid Replaced-Destination node pairs.

The heuristic described aims to promote Test Program feasibility by favouring the reuse of feasible portions of the MCT. The Test Program depicted in Figure 8.2c throws a runtime exception at instruction 6; the feasible portion of this program is thus the subsequence of instructions 1 to 5, whereas instructions 6 and 7 form the unfeasible subsequence. These sequences can be mapped directly to MCS entries which, in turn, can be matched to the corresponding MCT node. The valid Replaced-Destination node pairs which fulfil the premise of the heuristic are, thus, the following: {0.1, 0.0.1}; {0.1.0, 0.0.0}; {0.1.0, 0.0}.

8.1.4 Method Call Tree Linearisation

Evaluating the quality of an individual involves its execution which, in turn requires decoding the MCT into the Test Program. However, if At-Nodes

Algorithm 8.1: Algorithm for Method Call Tree linearisation in the presence of At-Nodes.

```
Data: Method Call Tree
Result: Method Call Sequence

Global Variables:
Current Node ← Root Node;
isDestinationNode ← false;
Previous MIO ← null;
MCS ← empty sequence;

begin Function linearizeMCT(Current Node, isDestinationNode)
    if Current Node ≠ Root Node and isDestinationNode = false then
        Previous MIO ← get MIO from from Parent Node of Current Node;
    if Current Node is an instance of At-Node then
        Destination Node ← get Destination Node from At-Node;
        call linearizeMCT(Destination Node, true);
    else if Current Node is an instance of Standard Node then
        Current MIO ← get MIO from Current Node;
        if Previous MIO ≠ null then
            add Current MIO to Parameter Providers List of Previous MIO;
        if isDestinationNode = false then
            Child Nodes List ← get Child Nodes List from Current Node;
            foreach Child Node in Child Nodes List do
                call linearizeMCT(Child Node, false);
            add Current MIO to MCS;
```

exist, a depth-first traversal algorithm does not suffice to linearise a tree; the linearisation algorithm must take into account the fact that certain parameters are supplied not by that node's children, but rather by the node pointed at by an At-Node. The algorithm depicted in Figure 8.1 describes the polymorphic recursive function utilised to obtain a MCS with basis on a MCT in the presence of At-Nodes.

8.2 The Reverse Object-Reuse Operator

If a MCT contains At-Nodes, some standard evolutionary operators, such as Mutation and Crossover, require the tree to be analysed and possibly modified prior to their application. This necessity is related with the fact that these operators replace subtrees in the original individual by newly created trees (in the case of the former) or by a copy of another individual's subtree (in the case of the latter); however, if the subtrees truncated in the original individual contain Destination Nodes their elimination will

With Object Reuse	Without Object Reuse
Object Reuse Op. (25%)	Mutation Op. (34%)
Reverse Object Reuse Op. / Mutation Op. (25%)	Crossover Op. (33%)
Reverse Object Reuse Op. / Crossover Op. (25%)	Reproduction Op. (33%)
Reproduction Op. (25%)	

Table 8.1: Sources of individuals.

render the MCT inconsistent and disable the possibility of translating it to a syntactically correct Test Program.

The Reverse Object Reuse operator's task is precisely that of pre-processing the individuals to be provided to other well-established operators, thus avoiding their reformulation. It starts by indexing all the At-Nodes in a MCT, and then proceeds to replace each At-Node with a clone copy of the sub-tree rooted at its Destination node. The resulting MCT can then be provided to another evolutionary operator. That is, the Reverse Object Reuse operator's purpose it that of being the first component of a breeding pipeline and acting as a source of individuals; it selects individuals directly from the population (e.g., using Tournament Selection [Koz92]), and provides the (possibly) modified individual to the operator at the end of the breeding pipeline. This process is schematised in Figure 8.1 on page 113.

8.3 Experimental Studies

The Object Reuse methodology described was embedded into the *eCrash* tool for the Evolutionary Testing of Object-Oriented Java software, with the objective of assessing its impact on both the efficiency and the effectiveness of the evolutionary search.

The Java `TreeMap` (an implementation of Red-Black Tree [CLRS01]) and `Vector` classes of JDK 1.4 [Sun03] were used as Test Objects. Their selection is supported by the fact that they are container classes, which are a typical benchmark in Software Testing of Object-Oriented programs; Red-Black Trees, in particular, have been empirically shown to be the most difficult to test among containers programs [Arc09b]. As MUTs, the 5 most complex public methods (in terms of their Cyclomatic Complexity Number (CCN) [McC76]) of each class were selected.

For each MUT, 2 sets of 20 runs were executed. The Object Reuse and Reverse Object Reuse operators were included in the first, and excluded from the second; Table 1 depicts the sources of individuals selected for each

set of runs. The decision of selecting equal probabilities for the Mutation, Crossover and Reproduction operators is supported by previous experiments described in Chapter 5. The remaining evolutionary parameters were common to both sets, and were defined as follows: a single population of 25 individuals was used; the search stopped if an ideal individual was found or after 200 generations; the selection method was Tournament Selection [Koz92] with size 2; the tree builder algorithm was PTC2 [Luk00b], with the minimum and maximum tree depths being defined as 4 and 14. The *eCrash* tool was configured in accordance to the setup proposed in [RZRFdV09].

An additional set of 20 runs, in which all individuals were randomly generated using the PTC2 algorithm (with minimum and maximum tree depths of 4 and 14), was performed for comparison purposes; because no evolutionary operators were used, Object Reuse was absent from the process. This random search stopped if an ideal individual was found or after the generation of 5000 individuals. The results were included in Table 8.2 on page 122.

8.3.1 Results and Discussion

The results depicted in Table 8.2 show that, for both classes, a higher percentage of runs attaining full structural coverage was achieved when including the Object Reuse operator as a source (with the exception being the `putAll(Map)` method of the `TreeMap` class). An average success rate of 62% was achieved with Object Reuse, whereas only 42.5% of the runs were successful without it.

What's more, the impossibility of attaining full structural coverage for some of the methods tested is symptomatic of the way in which the lack of the Object Reuse functionality can hinder the evolutionary search. In fact, several search methods – in particular, `Vector`'s `indexOf` and `lastIndexOf`; and `TreeMap`'s `put`, `remove` and `get` – rely on `equals` to verify if an item is contained in a collection. This means that if instances are not reused, the search for non-`null` arguments of type `Object` will fail. A commonly used workaround (e.g., [RZRdV08a]) is that of including substitute classes into the Test Cluster, which extend `Object` and override `equals` with a less stringent implementation (e.g., `String`); this approach, however, does not suffice for the following reasons:

- certain test scenarios may specifically involve using classes that do not override `equals`, or the `Object` class itself.

- the decision on which additional classes to include into the Test Cluster is problem specific and human dependant; to the best of our knowledge, no systematic strategy has been proposed to automate this task.
- the inclusion of redundant classes into the Test Cluster will enlarge the search space and will thus have negative consequences on the efficiency of the search (cf. Chapter 7).

The graphs depicted in Figure 8.4 provide an overview of the way in which the runs evolved, and on how the Object Reuse methodology affects the Test Data generation process in terms of structural coverage (Figures 8.4a and 8.4d), Test Program size (Figures 8.4b and 8.4e) and feasibility (Figures 8.4c and 8.4f). The runs in which Object Reuse was employed yield solutions with shorter MCS length (a difference of 20.3%, on average, for `TreeMap`, and 12% for `Vector`). Also, feasibility is significantly promoted, with an average increase of 4% for both the `TreeMap` and `Vector` classes.

These observations show that the methodology proposed is not only able to enhance the effectiveness of the Test Case generation process, but also its efficiency:

- it yields solutions with smaller overall size and lower average structural complexity, thus contributing positively to the area of MCS minimisation. Simpler and shorter Test Programs do not only reduce the computational effort involved in compilation and execution; they also ease the (mostly human-dependant) task of defining a mechanism for checking that the output of a program is correct given some input (i.e., an oracle).
- The application of the Replaced-Destination Node Pair Selection heuristic is able to increase the average feasibility of the generated Test Programs. Because only feasible Test Programs are concluded with a call to the MUT, a higher level of feasibility will increase the performance of the Test Data generation process (cf. Chapter 5).

8.4 Related Work

The proposed approach to Object Reuse has some similarities with Koza's work on Automatically Defined Functions (ADFs) [Koz94]. ADFs enable GP to solve a variety of problems in a way that can be interpreted as a

MUT	CCN	With OR %f	With OR #i	W/out OR %f	W/out OR #i	Random %f	Random #i
`TreeMap`							
put(Object,Object)	10	10%	4563	0%	5000	0%	5000
putAll(Map)	10	85%	1389	95%	1154	75%	2385
remove(Object)	3	25%	4119	0%	5000	0%	5000
containsValue(Object)	3	100%	501	100%	548	100%	628
get(Object)	2	25%	4000	0%	5000	0%	5000
`Vector`							
lastIndexOf(Object,int)	10	60%	3203	0%	5000	0%	5000
indexOf(Object,int)	8	40%	4243	0%	5000	0%	5000
removeElementAt(int)	6	85%	1829	75%	2258	70%	2948
addAll(int,Collection)	5	100%	871	95%	1130	80%	1668
remove(int)	4	90%	1904	80%	2545	80%	2815

Table 8.2: Percentage of runs attaining full coverage (*%f*) and average number of individuals evaluated per run (*#i*); for the *With OR*, *Without OR* and *Random* runs; for the 5 public methods with the highest CCN of the `TreeMap` and `Vector` classes.

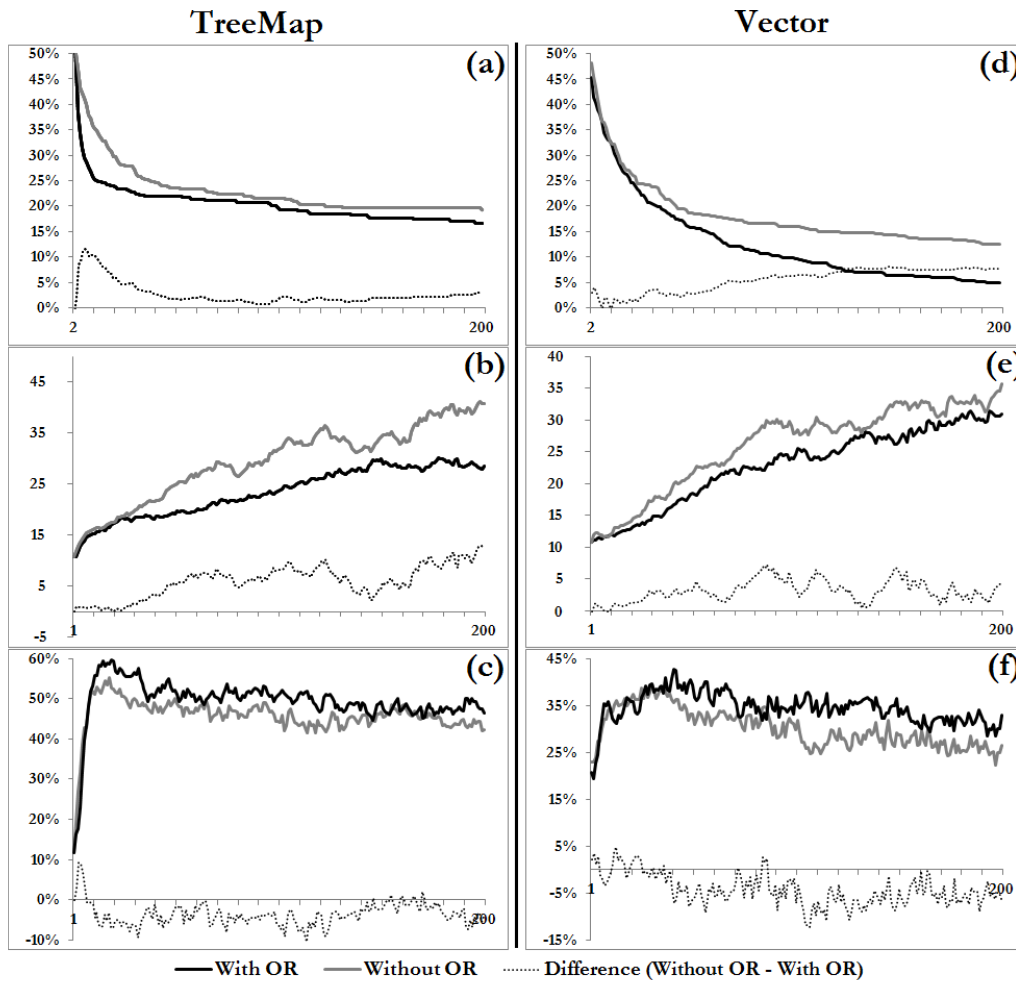

Figure 8.4: Average percentage of CFG nodes left to be covered per generation *(a and d)*, average MCS length per generation *(b and e)*, and average percentage of feasible individuals per generation *(c and f)*; for the *With OR* and *Without OR* runs; for the 5 public methods with the highest CCN of the `TreeMap` and `Vector` classes.

decomposition of a problem into subproblems, a solving of the subproblems, and an assembly of the solutions to the subproblems into a solution to the overall problem; an individual's genotype usually consists of a forest of trees (or functions), which are then called repeatedly from the main tree. Therefore, ADFs do allow function reuse, as the possibility of selecting and calling the same function multiple times exists. However, functions in Object-Oriented languages typically return object references, and each individual function call – even to the same function – returns a distinct reference. As such, ADFs do not enable Object Reuse, as the possibility of using the object reference returned by a single function call more than once is not possible.

The Object Reuse methodology described also shares some characteristics with graph-based approaches to GP, such as Parallel Distributed Genetic Programming (PDGP) [Pol97] and Cartesian Genetic Programming (CGP) [MT00], as it also involves loosening the interpretation of the edges of a MCT thus effectively transforming it into a graph. However, to the best of the authors' knowledge, there has been no research on applying any of the above approaches to the generation of Object-Oriented software and, in particular, to Object-Oriented Evolutionary Testing; conversely, STGP has been extended to support type inheritance and polymorphism [HSW96, Yu01b], and extensive work has been performed on applying it to Object-Oriented Evolutionary Testing (cf. Chapter 3). As such, we believe that the methodology proposed constitutes a significant novel contribution to the area.

The only previous approach to Object Reuse known to the author does not involve a loosening of the interpretation of the edges of a MCT, but rather a loosening of the parameter object assignments during tree linearisation. In [Wap07], Wappler proposes employing an *Object Pool* that stores references to all the objects created during a Test Program execution; this pool is consulted if a parameter object is required for a method call, and a parameter object selector component selects the instance to be used among all available instances of the required type (e.g., Listing 8.2).

There are, however, some drawbacks to the Object Pool approach to Object Reuse. Firstly, all the objects, even those that are not used, must be created and stored in the Object Pool, which will obviously increase the length and complexity of Test Programs. Also, and perhaps most importantly, changing parameter object assignments during tree linearisation will result in a discrepancy between the individual's hereditary information (i.e., its genotype) and its actual observed properties (i.e., its phenotype); in

```
Vector vector1 = new Vector();
ObjectPool.addInstance(vector1);
Object object2 = new Object();
ObjectPool.addInstance(object2);
Vector poolVector1 = ObjectPool.getInstance(Vector.class);
Object poolObject2 = ObjectPool.getInstance(Object.class);
boolean boolean3 = poolVector1.add(poolObject2);
ObjectPool.addInstance(boolean3);
Object object4 = new Object();
ObjectPool.addInstance(object4);
Vector poolVector3 = ObjectPool.getInstance(Vector.class);
Object poolObject4 = ObjectPool.getInstance(Object.class);
int int5 = poolVector3.indexOf(poolObject4);
```

Listing 8.2: Example Test Program employing the Object Pool approach to Object Reuse [Wap07].

other words, the Test Program might not directly correspond to the MCT. Considering that an individual's evaluation is performed at the phenotype level, the Test Program must be an exact translation of the MCT in order for the fitness to be accurately assessed and reflect an individual's quality.

8.5 Summary

The goal of Object-Oriented Evolutionary Testing is to find a set of Test Cases that satisfies a certain test criterion. If structural adequacy criteria are employed, Object Reuse is a feature of paramount importance, as it enables the generation of Test Programs that exercise specific structures of software that would not be reachable otherwise.

Object Reuse means that one instance can be passed to multiple methods as an argument, or multiple times to the same method as arguments; the main contribution of this work is that of proposing a novel methodology which enables Object Reuse on typed GP-based approaches to Object-Oriented Evolutionary Testing. The proposed approach to Object Reuse involves the definition of novel type of GP nodes – the At-Nodes – that "point to" other nodes, thus effectively enabling the reuse of portions of the tree and, specifically, the reuse of the object references returned by the functions corresponding to the reused sub-trees. Additionally, At-Nodes may be removed from a tree; this particular feature allows avoiding the reformulation of other well-established evolutionary operators, such as Mutation and Crossover.

Besides enhancing the effectiveness of the search, the experimental studies performed show that the proposed methodology is able to increase the performance of the Test Data generation process: it yields solutions with smaller overall size and lower structural complexity, and it is able to increase the average feasibility of Test Programs by means of a specific heuristic for the selection of the nodes to be used by the Object Reuse operator.

Chapter 9

Conclusions and Future Work

This Thesis presented a Genetic Programming-based approach to the generation of structural Unit Test data for Object-Oriented software. Relevant contributions include (but are not limited to) the introduction of novel methodologies for search guidance, Input Domain Reduction, constraint selection, Object Reuse – and the presentation of the *eCrash* Test Data generation tool.

Test Data generation is, in fact, a difficult research topic, especially if the goal is to implement an automated, adaptable and inexpensive solution. The State Problem of Object-Oriented programs, in particular, requires the definition of methodologies that promote the coverage of problematic structures and difficult control-flow paths. We proposed tackling this particular challenge by defining weighted Control-Flow Graph nodes and constantly adapting the direction of the search. This strategy also causes the fitness of feasible Test Programs to fluctuate throughout the search process, and allows unfeasible Test Programs to be considered at certain points of the evolutionary search – namely, once the feasible ones cease to be interesting because they exercise recurrently traversed structures.

An Input Domain Reduction methodology for eliminating irrelevant variables from Object-Oriented Evolutionary Testing problems was also proposed; it is based on the concept of Purity Analysis, and provides a means to automatically identify and remove Function Set entries that do not contribute to the definition of interesting test scenarios. This process also ensures that Test Programs are not rendered unfeasible by the inclusion of uninteresting instructions, which makes it particularly important in the context of our approach, given that the Test Data evaluation strategy defined

considers unfeasible Test Programs at certain stages of the search.

The inclusion of relevant instructions into the generated Test Programs (by means of Mutation) also provides the rationale for the adaptive methodology presented for dynamically updating the selection probabilities of the constraints defined in the Function Set. This strategy obtains feedback from the solutions previously produced and evaluated in order to promote diversity and Test Program feasibility.

The Object Reuse strategy presented enables the generation of Test Programs that exercise specific structures of software that would not be reachable otherwise; it is thus especially important if White-Box adequacy criteria are employed. Object Reuse means that one instance can be passed to multiple methods as an argument, or multiple times to the same method as arguments; the proposed approach to Object Reuse involves the definition of novel type of Genetic Programming nodes that “point to” other nodes, thus effectively enabling the reuse of the object references returned by the functions corresponding to the reused sub-trees. Additionally, At-Nodes may be removed from a tree; this particular feature allows avoiding the reformulation of other well-established evolutionary operators, such as Mutation and Crossover.

The initial objectives of this work were those of positively contributing to improve the level of automation and performance of the (often neglected but enormously important) Software Testing process, and of investigating the pertinence of applying Evolutionary Algorithms to Test Data generation problems. We believe that it is possible to affirm that this goal was achieved, not only as a result of the proposals made to enhance both the efficiency and the effectiveness of Object-Oriented Evolutionary Testing approaches, but also because this research resulted in the development of an automated tool which demonstrates the applicability of Genetic Programming to this purpose.

The *eCrash* tool embodies the approach to Evolutionary Testing proposed; it was implemented during the course of the research presented this Thesis, and allowed studying and experimenting with novel methodologies for improving the Evolutionary Testing process. Even though it is currently in a prototype development stage, it is fully functional and is applicable to a vast array of Object-Oriented Test Objects. Nevertheless, we are actively working on developing a stable, user-friendly and well-documented implementation of this framework; near-future plans involve publishing an IDE-integrated version of *eCrash*, that can be used by Software Testers in a production environment and Evolutionary Testing researchers alike.

We also plan to address some of the limitations which we still did not have the chance to investigate: the testing and structural coverage of non-public methods (via an object's public interface) is an important issue. Future work will also be focused on addressing the challenges posed by the three cornerstones of Object-Oriented programming: Encapsulation, Inheritance, and Polymorphism.

The importance of the Inheritance and Polymorphism properties, in particular, is yet to be fully studied by researchers in this area. Inheritance allows the treatment of an object as its own type or its base type; Polymorphism means "different forms", and allows one type to express its distinction from another similar type through differences in behaviour of the methods that can be called through the base class [Eck02]. *Search space sampling* deals with the inclusion of *all* the relevant variables to a given Test Object into the Test Data generation problem, so as to enable the coverage of the entire search space whenever possible and improve the effectiveness the approach. Because the Test Cluster cannot possibly include all the subclasses that may override the behaviours of the classes which are relevant for the Test Object, adequate strategies for search space sampling – which take the commonality among classes and their relationships with each other into account – are of paramount importance. Steps in this direction have already been taken as a result of the proposal of the Adaptive Evolutionary Testing methodology described in Chapter 7; static analysis methodologies will be considered for sampling the search space and also for defining the initial constraint selection probabilities.

Future research plans also involve addressing the oracle generation problem, and investigating the possibility of automating a mechanism for checking if the output of a program is correct given some input; in fact, the frequent non-existence of an oracle threatens to undo much of the progress made in automating Test Data generation, as a human tester is still required to perform this task manually. Research questions include verifying to what extent Weyuker's statement labelling some programs as "non-testable" [Wey82] due to the difficulty of automatically generating a test oracle for such programs is true, and exploring the possibility of circumventing the oracle generation problem by means of pseudo-oracles [DW81, McM09].

We are also planning to experiment with parallel systems in order to enhance our methodology's performance. The breeding and fitness calculation procedures, in particular, are inherently parallelisable; all the individuals in population my be created and evaluated simultaneously. Additionally, we intend to treat the Test Data generation and selection process as a Multi

Objective Optimization problem [YH07], so as to take into account several goals simultaneously (e.g, structural coverage, Test Program length, Test Set size, execution cost).

Bibliography

[ABHPW08] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege. A systematic review of the application and empirical investigation of evolutionary testing. Technical Report Simula Technical Report Simula.SE.293, 2008. [cited at p. 27]

[ABHPW09] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering*, 99(1), 2009. [cited at p. 14, 48, 49]

[Ang95] Peter J. Angeline. Adaptive and self-adaptive evolutionary computations. In *Computational Intelligence: A Dynamic Systems Perspective*, pages 152–163. IEEE Press, 1995. [cited at p. 97, 98]

[Arc08] Andrea Arcuri. On the automation of fixing software bugs. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 1003–1006, New York, NY, USA, 2008. ACM. [cited at p. 2, 45]

[Arc09a] Andrea Arcuri. *Automatic software generation and improvement through search based techniques.* PhD thesis, University of Birmingham, 2009. [cited at p. 48]

[Arc09b] Andrea Arcuri. Insight knowledge in search based software testing. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1649–1656, New York, NY, USA, 2009. ACM. [cited at p. 47, 119]

[AWCY08] Andrea Arcuri, David Robert White, John Clark, and Xin Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *Proceedings of the 7th International Conference on Simulated Evolution And Learning (SEAL '08)*, pages 61–70. Springer, 2008. [cited at p. 45]

[AY07a] Andrea Arcuri and Xin Yao. Coevolving programs and unit tests from their specification. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 397–400, New York, NY, USA, 2007. ACM. [cited at p. 2, 44, 50]

[AY07b] Andrea Arcuri and Xin Yao. A memetic algorithm for test data generation of object-oriented software. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC)*, pages 2048–2055. IEEE, 2007. [cited at p. 2, 46, 74, 84]

[AY07c] Andrea Arcuri and Xin Yao. On test data generation of object-oriented software. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 72–76, Washington, DC, USA, 2007. IEEE Computer Society. [cited at p. 45, 50]

[AY07d] 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. [cited at p. 46]

[AY08a] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '08)*, pages 162–168. IEEE Computer Society, 2008. [cited at p. 45]

[AY08b] Andrea Arcuri and Xin Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008. [cited at p. 28, 33, 47]

[Bal02] Francesco Balena. *Programming Microsoft Visual Basic .NET (Core Reference)*. Microsoft Press, Redmond, WA, USA, 2002. [cited at p. 5]

[BDMN79] G.M. Birtwhistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Chartwell-Bratt Ltd, 1979. [cited at p. 6]

[Bei90] Boris Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990. [cited at p. 1, 13, 14, 16]

[Ber02] Hans Bergsten. *Javaserver Pages*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. [cited at p. 7]

[Ber07] 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. [cited at p. 11, 27, 28]

[BFM97] Thomas Back, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Bristol, UK, UK, 1997. [cited at p. 1, 97]

[BL05] C.J. Burgess and M. Lefley. *Can Genetic Programming improve Software Effort Estimation? A Comparative Evaluation*, volume 16, pages 95–105. World Scientific Publishing Co., May 2005. [cited at p. 45]

[BM09] William Bateson and Gregor Mendel. *Mendel's principles of heredity,*. Cambridge [Eng.]University Press,, 1909. http://www.biodiversitylibrary.org/bibliography/1057. [cited at p. 19]

[BME+07] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison-Wesley Professional, 3 edition, April 2007. [cited at p. 5, 7, 8, 10]

[BMH06] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0 (5th Edition)*. O'Reilly Media, Inc., May 2006. [cited at p. 7]

[BS94] 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. [cited at p. 28]

[BS02] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies –a comprehensive introduction. *Natural Computing: an international journal*, 1(1):3–52, 2002. [cited at p. 19]

[CK06a] 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. [cited at p. 39]

[CK06b] 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. [cited at p. 50]

[CKP05] Yoonsik Cheon, Myoung Kim, and Ashaveena Perumandla. A complete automation of unit testing for java programs. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the International Conference on Software Engineering Research and Practice, SERP 2005, Las Vegas, Nevada, USA, June 27-29, 2005, Volume 1*, pages 290–295. CSREA Press, 2005. [cited at p. 2, 39]

[CLRS01] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001. [cited at p. 45, 119]

[Dar95] Charles Darwin. *The Origin of Species*. Gramercy, May 1995. [cited at p. 19]

[dCT03] Leandro Nunes de Castro and Jon Timmis. Artificial immune systems as a novel soft computing paradigm. *Soft Comput.*, 7(8):526–544, 2003. [cited at p. 18, 46]

[dCZ02] Leandro Nunes de Castro and Fernando J. Von Zuben. Learning and optimization using the clonal selection principle. *IEEE Trans. Evolutionary Computation*, 6(3):239–251, 2002. [cited at p. 46]

[DJAR07] C. S. Siva Dharsana, D. Nithila Jennifer, A. Askarunisha, and N. Ramaraj. Java based test case generation and optimization using evolutionary testing. In *ICCIMA '07: Proceedings of the International Conference on Computational Intelligence and Multimedia Applications (ICCIMA 2007)*, pages 44–49, Washington, DC, USA, 2007. IEEE Computer Society. [cited at p. 2, 40]

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization (Bradford Books)*. The MIT Press, July 2004. [cited at p. 45]

[dV01] Francisco Fernández de Vega. *Distributed Genetic Programming Models with Application to Logic Synthesis on FPGAs*. PhD thesis, University of Extremadura, 2001. [cited at p. 19]

[DW81] Martin D. Davis and Elaine J. Weyuker. Pseudo-oracles for non-testable programs. In *ACM 81: Proceedings of the ACM '81 conference*, pages 254–257, New York, NY, USA, 1981. ACM. [cited at p. 48, 129]

[Eck02] Bruce Eckel. *Thinking in Java*. Prentice Hall Professional Technical Reference, 2002. [cited at p. 9, 129]

[Edm01] Bruce Edmonds. Meta-genetic programming: Co-evolving the operators of variation. *Elektrik*, 9(1):13–29, May 2001. Turkish Journal Electrical Engineering and Computer Sciences. [cited at p. 99]

[EKdCA98] Matthew Evett, Taghi Khoshgoftar, Pei der Chien, and Edward Allen. Gp-based software quality prediction. In *in: Proc. 3rd Annual Genetic Programming Conference*, 1998. [cited at p. 45]

[Ern03] 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. [cited at p. 16]

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. [cited at p. 5, 6]

[FCA09] Javier Ferrer, Francisco Chicano, and Enrique Alba. Dealing with inheritance in oo evolutionary testing. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1665–1672, New York, NY, USA, 2009. ACM. [cited at p. 41]

[fITS98] International Committee for Information Technology Standards. Programming languages – smalltalk. Technical Report ANSI/INCITS 319-1998, 1 1998. [cited at p. 6]

[Fog62] L. J. Fogel. Toward inductive inference automata. In *IFIP Congress*, pages 395–400, 1962. [cited at p. 19]

[Fog99] Lawrence J. Fogel. *Intelligence through simulated evolution: forty years of evolutionary programming*. John Wiley & Sons, Inc., New York, NY, USA, 1999. [cited at p. 19]

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. [cited at p. 9]

[Glo89] Fred Glover. Tabu search–Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989. [cited at p. 19]

[Gol89] D.E Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, 1989. [cited at p. 19]

[GR08] Nirmal Kumar Gupta and Mukesh Kumar Rohil. Using genetic algorithm for unit testing of object oriented software. In *ICETET '08: Proceedings of the 2008 First International Conference on Emerging Trends in Engineering and Technology*, pages 308–313, Washington, DC, USA, 2008. IEEE Computer Society. [cited at p. 2, 44, 50]

[Har07a] Mark Harman. Automated test data generation using search based software engineering. In *AST '07: Proceedings of the Second International Workshop on Automation of Software Test*, page 2, Washington, DC, USA, 2007. IEEE Computer Society. [cited at p. 1]

[Har07b] 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. [cited at p. 2, 27]

[HHH+04] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Trans. Softw. Eng.*, 30(1):3–16, 2004. [cited at p. 47]

[HHL+07] 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. [cited at p. 28, 47, 81, 84]

[Hil90] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Phys. D*, 42(1-3):228–234, 1990. [cited at p. 44]

[Hir67] I. N. Hirsch. Memmap/360. Technical Report TR P-1168, IBM Systems Development Division, Product Test Laboratories, Poughkeepsie, N.Y., 2 1967. [cited at p. 16]

[HME97] Robert Hinterding, Zbigniew Michalewicz, and A. E. Eiben. Adaptation in evolutionary computation: A survey. In *In Proceedings of the Fourth International Conference on Evolutionary Computation (ICEC 97*, pages 65–69. IEEE Press, 1997. [cited at p. 97, 98, 100]

[HMZ09] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, 2009. [cited at p. 2, 27, 29, 45, 48, 49]

[Hol62] John H. Holland. Outline for a logical theory of adaptive systems. *J. ACM*, 9(3):297–314, 1962. [cited at p. 19]

[Hol92] 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. [cited at p. 21]

[HSW96] 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. [cited at p. 26, 124]

[IEE87] IEEE. Ansi/ieee std 1008-1987: Ieee standard for software unit testing, 1987. [cited at p. 16]

[Ind09] The TIOBE Programming Community Index. www.tiobe.com/tiobe_index/index.htm, 11 2009. [cited at p. 6, 7]

[IX07] Kobi Inkumsah and Tao Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 425–428, November 2007. [cited at p. 2, 39]

[IX08] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, September 2008. [cited at p. 2, 40]

[KDR06] 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. [cited at p. 18]

[KDR07] Alex Kinneer, Matthew B. Dwyer, and Gregg Rothermel. Sofya: Supporting rapid development of dynamic program analyses for java. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 51–52, Washington, DC, USA, 2007. IEEE Computer Society. [cited at p. 55, 70]

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. [cited at p. 43]

[KJS98] 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. [cited at p. 60]

[KL86] Pekka J. Korhonen and Jukka Laakso. A visual interactive method for solving the multiple criteria problem. *European Journal of Operational Research*, 24(2):277–287, February 1986. [cited at p. 18]

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, December 1992. [cited at p. 19, 21, 22, 75, 101, 112, 119, 120]

[Koz94] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press, Cambridge, Massachusetts, 1994. [cited at p. 25, 121]

[KP08a] Gal Katz and Doron Peled. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *ATVA '08: Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, pages 33–47, Berlin, Heidelberg, 2008. Springer-Verlag. [cited at p. 45]

[KP08b] Gal Katz and Doron Peled. Model checking-based genetic programming with an application to mutual exclusion. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 141–156. Springer, 2008. [cited at p. 45]

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language (2nd Edition)*. Prentice Hall PTR, 2 edition, April 1988. [cited at p. 6]

[LA87] P. J. M. Laarhoven and E. H. L. Aarts, editors. *Simulated annealing: theory and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987. [cited at p. 18, 38]

[LB03] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, 2003. [cited at p. 12]

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006. [cited at p. 39, 82]

[Lis87] Barbara Liskov. Data Abstraction and Hierarchy. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Addendum to the Proceedings*, volume 23, pages 17–34, October 1987. [cited at p. 10]

[LL02] P. Larranaga and J.A. Lozano. *Estimation of distribution algorithms: A new tool for evolutionary computation*. Kluwer Academic Pub, 2002. [cited at p. 45]

[LR07] K. Liaskos and M. Roper. Automatic test-data generation: An immunological approach. In *Testing: Accademic and Industrial Conference - Practice and Research Techniques (TAIC PART)*, pages 77–81. IEEE Computer Society, September 2007. [cited at p. 46]

[LR08] Konstantinos Liaskos and Marc Roper. Hybridizing evolutionary testing with artificial immune systems and local search. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 211–220, Washington, DC, USA, 2008. IEEE Computer Society. [cited at p. 2, 46]

[LRW07] Konstantinos Liaskos, Marc Roper, and Murray Wood. Investigating dataflow coverage of classes using evolutionary algorithms. In *GECCO '07:*

Proceedings of the 9th annual conference on Genetic and evolutionary computation, pages 1140–1140, New York, NY, USA, 2007. ACM. [cited at p. 2, 46]

[Luk00a] Sean Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat.* PhD thesis, Department of Computer Science, University of Maryland, A. V. Williams Building, University of Maryland, College Park, MD 20742 USA, 2000. [cited at p. 102]

[Luk00b] Sean Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, September 2000. [cited at p. 102, 120]

[Luk09a] Sean Luke. ECJ 19: A Java evolutionary computation library. http://cs.gmu.edu/∼eclab/projects/ecj/, 2009. [cited at p. 55, 70]

[Luk09b] Sean Luke. *Essentials of Metaheuristics.* 2009. available at http://cs.gmu.edu/∼sean/book/metaheuristics/. [cited at p. 21, 23]

[LW04] Kanglin Li and Mengqi Wu. *Effective Software Test Automation: Developing an Automated Software Testing Tool.* SYBEX Inc., Alameda, CA, USA, 2004. [cited at p. 10, 11, 12]

[LWL05] 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. [cited at p. 2, 45]

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. [cited at p. 7, 58]

[MA05] Timo Mantere and Jarmo T. Alander. Evolutionary software engineering, a review. *Appl. Soft Comput.*, 5(3):315–331, 2005. [cited at p. 2, 48, 49]

[Mar94] John J. Marciniak, editor. *Encyclopedia of software engineering.* Wiley-Interscience, New York, NY, USA, 1994. [cited at p. 10, 11]

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976. [cited at p. 119]

[McM04] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004. [cited at p. 1, 2, 27, 48, 49]

[McM09] Phil McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1689–1696, New York, NY, USA, 2009. ACM. [cited at p. 48, 129]

[MH03] P. McMinn and M. Holcombe. The state problem for evolutionary testing, 2003. [cited at p. 29]

[Mic94] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (2nd, extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1994. [cited at p. 20]

[Mon93] David J. Montana. Strongly typed genetic programming. Technical Report #7866, 10 Moulton Street, Cambridge, MA 02138, USA, 7 1993. [cited at p. 26]

[Mon95] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995. [cited at p. 2, 26, 33, 52]

[Mos89] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech Concurrent Computation Program, C3P Report*, 826:1989, 1989. [cited at p. 46]

[MS76] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Softw. Eng.*, 2(3):223–226, 1976. [cited at p. 37]

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. [cited at p. 15]

[MT00] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In *Proceedings of the European Conference on Genetic Programming*, pages 121–132, London, UK, 2000. Springer-Verlag. [cited at p. 124]

[NL03] Colin J. Neill and Phillip A. Laplante. Requirements engineering: The state of the practice. *IEEE Software*, 20(6):40–45, 2003. [cited at p. 12]

[Ols94] J. R. Olsson. Inductive functional programming using incremental program transformation and execution of logic programs by iterative-deepening a* sld-tree search. research report 189, University of Oslo, 1994. Dr. scient. thesis, edited version. [cited at p. 26]

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, New York, NY, USA, 2007. ACM. [cited at p. 40]

[PLM08] Riccardo Poli, William B. Langdon, and Nicholas F. Mcphee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, March 2008. [cited at p. 21, 23, 24, 25, 26]

[Pol97] Riccardo Poli. Evolution of graph-like programs with parallel distributed genetic programming. In Thomas Bäck, editor, *ICGA*, pages 346–353. Morgan Kaufmann, 1997. [cited at p. 124]

[Raj04] R. Rajendran. White paper on unit testing. Online at: http://www.mobilein.com/WhitePaperonUnitTesting.pdf, 2004. [cited at p. 11, 15]

[RdVZR07] José Carlos Bregieiro Ribeiro, Francisco Fernández 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 at the 25th Brazilian Symposium on Computer Networks and Distributed Systems*, pages 143–156. Brazilian Computer Society (SBC), 2007. [cited at p. 43]

[Rec65] I. Rechenberg. Cybernetic solution path of an experimental problem. Technical report, Royal Air Force Establishment, 1965. [cited at p. 19]

[Rib08] José Carlos Bregieiro Ribeiro. Search-based test case generation for object-oriented java software using strongly-typed genetic programming. In *GECCO '08: Proceedings of the 2008 GECCO Conference Companion on Genetic and Evolutionary Computation*, pages 1819–1822, New York, NY, USA, 7 2008. ACM. [cited at p. 43]

[RNC+96] Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. [cited at p. 38]

[Roy87] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. [cited at p. 12]

[Run06] Per Runeson. A survey of unit testing practices. *IEEE Softw.*, 23(4):22–29, 2006. [cited at p. 15]

[Rut08] Leszek Rutkowski. *Computational Intelligence: Methods and Techniques*. Springer Publishing Company, Incorporated, 2008. [cited at p. 21]

[RZdV07a] José Carlos Bregieiro Ribeiro, Mário Zenha-Rela, and Francisco Fernández de Vega. ecrash: a framework for performing evolutionary testing on third-party java components. In *CEDI JAEM'07: Proceedings of the I Jornadas sobre Algoritmos Evolutivos y Metaheuristicas at the II Congreso Español de Informática*, pages 137–144, 2007. [cited at p. 43]

[RZdV07b] José Carlos Bregieiro Ribeiro, Mário Zenha-Rela, and Francisco Fernández de Vega. An evolutionary approach for performing structural unit-testing on third-party object-oriented java software. In *Nature Inspired Cooperative Strategies for Optimization (NICSO 2007)*, volume 129/2008 of *Studies in Computational Intelligence*, pages 379–388. Springer Berlin / Heidelberg, 11 2007. [cited at p. 43]

[RZRdV08a] José Carlos Bregieiro Ribeiro, Mário Zenha-Rela, and Francisco Fernández de Vega. A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software. In *AST '08: Proceedings of the 3rd International Workshop on Automation of Software Test*, pages 85–92, New York, NY, USA, 2008. ACM. [cited at p. 43, 120]

[RZRdV08b] José Carlos Bregieiro Ribeiro, Mário Alberto Zenha-Rela, and Francisco Fernández de Vega. Strongly-typed genetic programming and purity analysis: input domain reduction for evolutionary testing problems. In *GECCO '08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, pages 1783–1784, New York, NY, USA, 7 2008. ACM. [cited at p. 43]

[RZRdV10a] José Carlos Bregieiro Ribeiro, Mário Alberto Zenha-Rela, and Francisco Fernández de Vega. Adaptive evolutionary testing: an adaptive approach to search-based test case generation for object-oriented software. In *NICSO 2010 - International Workshop on Nature Inspired Cooperative Strategies for Optimization (to appear)*, Studies in Computational Intelligence. Springer, 5 2010. [cited at p. 43]

[RZRdV10b] José Carlos Bregieiro Ribeiro, Mário Alberto Zenha-Rela, and Francisco Fernández de Vega. Enabling object reuse on genetic programming-based approaches to object-oriented evolutionary testing. In *EuroGP 2010 - 13th European Conference on Genetic Programming (to appear)*, Lecture Notes in Computer Science. Springer, 4 2010. [cited at p. 43]

[RZRFdV09] José Carlos Bregieiro Ribeiro, Mário Alberto Zenha-Rela, and Francisco Fernández de Vega. Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. *Inf. Softw. Technol.*, 51(11):1534–1548, 2009. [cited at p. 43, 120]

[SA06] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006. [cited at p. 39, 40]

[SAY07] 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. [cited at p. 2, 45, 72]

[SD07] S. N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms*. Springer Publishing Company, Incorporated, 2007. [cited at p. 33]

[See06] Arjan Seesing. Evotest: Test case generation using genetic programming and software analysis. Master's thesis, Delft University of Technology, 2006. [cited at p. 2, 41, 44]

[Sen07] Koushik Sen. Concolic testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, New York, NY, USA, 2007. ACM. [cited at p. 39]

[SG06] Arjan Seesing and Hans-Gerhard Gross. A genetic programming approach to automated test generation for object-oriented software. *International Transactions on System Science and Applications*, 1(2):127–134, 2006. [cited at p. 2, 41, 43, 50, 84]

[SR04] A. Salcianu and M. Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAILTR-949, MIT, May 2004. [cited at p. 62, 81, 86]

[SR05] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for java programs. In *VMCAI'05: Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2005. [cited at p. 81, 82, 86]

[Sun03] Sun Microsystems. *JavaTM 2 Platform, Standard Edition, v 1.4.2, API Specification*, 2003. http://java.sun.com/j2se/1.4.2/docs/api/. [cited at p. 5, 7, 111, 119]

[Tal09] El-Ghazali Talbi. *Metaheuristics : from design to implementation.* John Wiley & Sons, 2009. [cited at p. 18]

[Tas02] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002. [cited at p. 10, 11, 13]

[TCMM02] Nigel Tracey, John Clark, John McDermid, and Keith Mander. A search-based automated test-data generation framework for safety-critical systems. *Systems engineering for business process change: new directions*, pages 174–213, 2002. [cited at p. 1]

[Tho89] D. Thomas. What's in an object? (object-oriented programming). 14(3):231–232, 234–236, 238, 240, 270–271, March 1989. [cited at p. 10]

[Til08] Pexwhite box test generation for .net. pages 134–153. 2008. [cited at p. 40]

[Ton04] 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. [cited at p. 1, 2, 27, 28, 37, 38, 39, 43, 46, 48, 60]

[Top02] Kim Topley. *J2ME in a nutshell.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. [cited at p. 7]

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, New York, 1993. [cited at p. 43]

[VDMW06] 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. [cited at p. 8, 16, 58]

[VPP06] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for java containers using state matching. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 37–48, New York, NY, USA, 2006. ACM. [cited at p. 47]

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999. [cited at p. 86]

[VTFD07] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In Erik Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 54–78. Springer, 2007. [cited at p. 112]

[Wap07] Stefan Wappler. *Automatic Generation Of Object-Oriented Unit Tests Using Genetic Programming*. PhD thesis, Technischen Universitat Berlin, 12 2007. [cited at p. 2, 26, 29, 43, 52, 59, 100, 111, 124, 125]

[WBS01] Joachim Wegener, Andr Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, 2001. [cited at p. 47]

[Wei08] Matt Weisfeld. *The Object-Oriented Thought Process*. Addison-Wesley Professional, 2008. [cited at p. 6]

[Wey82] Elaine J. Weyuker. On testing non-testable programs. *Comput. J.*, 25(4):465–470, 1982. [cited at p. 129]

[Wil02] Mickey Williams. *Microsoft Visual C# (Core Reference)*. Microsoft Press, Redmond, WA, USA, 2002. [cited at p. 5]

[WL05] 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. [cited at p. 29, 38, 43]

[WM97] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. [cited at p. 19]

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206, New York, NY, USA, 1999. ACM. [cited at p. 82]

[WS07] Stefan Wappler and Ina Schieferdecker. Improving evolutionary class testing in the presence of non-public methods. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 381–384, New York, NY, USA, 2007. ACM. [cited at p. 2, 42, 50]

[WW06a] 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. [cited at p. 2, 31, 41, 42]

[WW06b] 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. [cited at p. 2, 14, 27, 41, 42, 62]

[XES+92] 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. [cited at p. 27, 37]

[XPV07] 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. [cited at p. 81, 82]

[XTdHS08] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Method-sequence exploration for automated unit testing of object-oriented programs. In *Proc. Workshop on State-Space Exploration for Automated Testing (SSEAT 2008)*, July 2008. [cited at p. 2, 40, 50]

[YH07] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150, New York, NY, USA, 2007. ACM. [cited at p. 45, 130]

[Yu01a] Tina Yu. Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. *Genetic Programming and Evolvable Machines*, 2(4):345–380, December 2001. [cited at p. 26]

[Yu01b] Tina Yu. Polymorphism and genetic programming. In *EuroGP '01: Proceedings of the 4th European Conference on Genetic Programming*, pages 218–233, London, UK, 2001. Springer-Verlag. [cited at p. 124]

[ZHR+06] Sharon Zakhour, Scott Hommel, Jacob Royal, Isaac Rabinovitch, Tom Risser, and Mark Hoeber. *The Java Tutorial: A Short Course on the Basics, 4th Edition (Java Series)*. Prentice Hall PTR, 4th edition, October 2006. [cited at p. 8, 59]

[ZLXJ03] Weicai Zhong, Jing Liu, Mingzhi Xue, and Licheng Jiao. Global numerical optimization using multi-agent genetic algorithm. In *ICCIMA '03: Proceedings of the 5th International Conference on Computational Intelligence and Multimedia Applications*, page 165, Washington, DC, USA, 2003. IEEE Computer Society. [cited at p. 45]

Appendices

Appendix A

Publications

The work presented in this Thesis is original work undertaken between June 2006 and June 2010 at the University of Extremadura, Spain. Portions of this work have been published elsewhere.

Journal Proceedings (ISI)

- Ribeiro, J. and Rela, M. Z. and Vega, F.F., “Test Case Evaluation and Input Domain Reduction strategies for the Evolutionary Testing of Object-Oriented software”. *Jornal of Information and Software Technology*, Volume 51, Issue 11, pp. 1534-1548. Elsevier, Butterworth-Heinemann, Newton, MA, USA, November 2009. ISSN: 0950-5849
  DOI: `http://dx.doi.org/10.1016/j.infsof.2009.06.009`

Conference Proceedings

- Ribeiro, J. and Rela, M. Z. and Vega, F.F., ”Adaptive Evolutionary Testing: an Adaptive Approach to Search-Based Test Case Generation for Object-Oriented Software”. In Proc. of the *NICSO’10 – 4th International Workshop on Nature Inspired Cooperative Strategies for Optimization*, pp. 185-197, Vol. 284/2010, Studies in Computational Intelligence – Springer, Granada, Spain, May 2010
  DOI: `http://dx.doi.org/10.1007/978-3-642-12538-6_16`

- Ribeiro, J. and Rela, M. Z. and Vega, F.F., ”Enabling Object Reuse on Genetic Programming-based Approaches to Object-Oriented Evo-

lutionary Testing". In Proc. of the *EuroGP'10 – 13th European Conference on Genetic Programming*, pp. 220-231, Vol. 6021/2010, Lecture Notes in Computer Science – Springer, Instanbul, Turkey, April 2010
DOI: `http://dx.doi.org/10.1007/978-3-642-12148-7_19`

- Ribeiro, J. and Rela, M. Z. and Vega, F.F., "An Adaptive Strategy for Improving the Performance of Genetic Programming-based Approaches to Evolutionary Testing". In Proc. of the *GECCO'09 – 11th Annual Conference on Genetic and Evolutionary Computation*, pp. 1949-1950, ACM, Montréal, Québec, Canada, July 2009
DOI: `http://doi.acm.org/10.1145/1569901.1570253`

- Ribeiro, J. , "Search-Based Test Case Generation for Object-Oriented Java Software Using Strongly-Typed Genetic Programming". In Proc. of the *GECCO'08 – Graduate Student Workshop*, pp. 1819-1822, ACM, Atlanta, Georgia, USA, July 2008
DOI: `http://doi.acm.org/10.1145/1388969.1388979`

- Ribeiro, J. and Rela, M. Z. and Vega, F.F., "Strongly-Typed Genetic Programming and Purity Analysis: Input Domain Reduction for Evolutionary Testing Problems". In Proc. of the *GECCO'08 – 10th Annual Conference on Genetic and Evolutionary Computation*, pp. 1783-1784, Atlanta, Georgia, USA, July 2008
DOI: `http://doi.acm.org/10.1145/1389095.1389439`

- Ribeiro, J. and Rela, M. Z. and Vega, F.F., "A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software". In Proc. of the *AST'08 – 3rd International Workshop on Automation of Software Test at the 30th International Conference on Software Engineering*, pp. 85-92, ACM, Leipzig, Germany, May 2008.
DOI: `http://doi.acm.org/10.1145/1389095.1389439`

- Ribeiro, J. and Rela, M. Z. and Vega, F.F., "An Evolutionary Approach For Performing Structural Unit-Testing On Third-Party Object-Oriented Java Software". In Proc. of the *NICSO'07 – 2nd International Workshop on Nature Inspired Cooperative Strategies for Optimization*, pp. 379-388, Vol. 129/2008, Studies in Computational Intelligence – Springer, Acireale, Italy, November 2007.
DOI: `http://dx.doi.org/10.1007/978-3-540-78987-1_34`

- Ribeiro, J. and Rela, M. Z. and Vega, F.F., “eCrash: a Framework for Performing Evolutionary Testing on Third-Party Java Components”. In Proc. of the *JAEM’07 – I Jornadas sobre Algoritmos Evolutivos y Metaheuristicas at the II Congreso Espaol de Informtica*, pp. 137-144, Zaragoza, Spain, September 2007. ISBN: 978-84-9732-593-6.
  URL: `http://jcbribeiro.googlepages.com/jribeiro_jaem07.pdf`

- Ribeiro, J. and Luis, B.M. and Rela, M. Z. , “Error propagation monitoring on windows mobile-based devices”. In Proc. of the *LADC’07 – 3rd Latin-American Symposium on Dependable Computing*, pp. 111-122, Vol. 4746/2007, Lecture Notes in Computer Science – Springer, Morelia, Mexico, September 2007.
  DOI: `http://dx.doi.org/10.1007/978-3-540-75294-3_9`

- Ribeiro, J. and Vega, F.F. and Rela, M. Z., “Using Dynamic Analysis of Java Bytecode for Evolutionary Object-Oriented Unit Testing”. In Proc. of the *WTF’08 – 8th Workshop on Testing and Fault Tolerance at the 25th Brazilian Symposium on Computer Networks and Distributed Systems*, pp. 143-156, Belém, Brazil, May 2007. ISBN: 85-766-0119-1.
  URL: `http://www.sbrc2007.ufpa.br/anais/2007/WTF04 - 02.pdf`

- Ribeiro, J. and Rela, M. Z. , “mCrash: a Framework for the Evaluation of Mobile Devices Trustworthiness Properties”. In Proc. of the *CSMU’06 – Conference on Mobile and Ubiquitous Systems*, pp. 163-166, Guimarães, Portugal, June 2006. ISBN: 972-8692-29-3.
  URL: `http://ubicomp.algoritmi.uminho.pt/csmu/proc/ribeiro-149.pdf`

Appendix B

Example ECJ Parameter and Function Files

```
# if otherwise noted, default Koza parameters are used
parent.0 = koza.params


### GENERAL PARAMETERS ###

# termination criteria
#
generations = 200
quit-on-run-complete = true

# subpopulation size
#
pop.subpop.0.size = 25

# problem
#
eval.problem = eCrash.MyProb
eval.problem.data = eCrash.MyGPData
eval.problem.stack.context.data = eCrash.MyGPData
eval.problem.size = 20
pop.subpop.0.species.ind = eCrash.MyGPIndividual

# BREEDING PIPELINES
#
pop.subpop.0.species.pipe = ec.breed.MultiBreedingPipeline
pop.subpop.0.species.pipe.generate-max = false
pop.subpop.0.species.pipe.num-sources = 3
pop.subpop.0.species.pipe.source.0 = ec.gp.koza.MutationPipeline
pop.subpop.0.species.pipe.source.0.prob = 0.34
pop.subpop.0.species.pipe.source.1 = ec.gp.koza.CrossoverPipeline
pop.subpop.0.species.pipe.source.1.prob = 0.33
pop.subpop.0.species.pipe.source.2 = ec.breed.ReproductionPipeline
pop.subpop.0.species.pipe.source.2.prob = 0.33
```

```
35
36 # SELECTION STRATEGY
37 #
38 gp.koza.mutate.source.0 = ec.select.TournamentSelection
39 gp.koza.mutate.ns.0 = ec.gp.koza.KozaNodeSelector
40 gp.koza.mutate.build.0 = ec.gp.build.PTC2
41 gp.koza.mutate.tries = 100
42 gp.koza.xover.source.0 = ec.select.TournamentSelection
43 gp.koza.xover.source.1 = same
44 gp.koza.xover.ns.0 = ec.gp.koza.KozaNodeSelector
45 gp.koza.xover.ns.1 = same
46 gp.koza.xover.maxdepth = 100
47 gp.koza.xover.tries = 100
48 breed.reproduce.source.0 = ec.select.TournamentSelection
49 breed.reproduce.tries = 100
50 select.tournament.size = 2
51
52 # TREE BUILDER
53 #
54 gp.tc.0.init = ec.gp.build.PTC2
55 gp.tc.1.init = ec.gp.build.PTC2
56 gp.build.ptc2.min-size = 4
57 gp.build.ptc2.max-size = 14
58 gp.build.ptc2.max-depth = 14
59 gp.fs.0 = ec.gp.build.PTCFunctionSet
60 gp.fs.0.name = f0
61
62
63 ### TEST OBJECT SPECIFIC PARAMETERS ###
64
65 # TREE INFO
66 #
67 pop.subpop.0.species.ind.numtrees = 1
68 pop.subpop.0.species.ind.tree.0  = ec.gp.GPTree
69 pop.subpop.0.species.ind.tree.0.tc = tc0
70 gp.tc.size = 1
71 gp.tc.0 = ec.gp.GPTreeConstraints
72 gp.tc.0.init = ec.gp.build.PTC2
73 gp.tc.0.name = tc0
74 gp.tc.0.fset = f0
75 gp.tc.0.returns = TREE
76
77 # ATOMIC TYPES
78 #
79 gp.type.a.size = 3
80 gp.type.a.0.name = classjavautilStack
81 gp.type.a.1.name = classjavalangObject
82 gp.type.a.2.name = TREE
83
84 # SET TYPES
85 #
86 gp.type.s.size = 1
87 gp.type.s.0.name = classjavalangObject-S
88 gp.type.s.0.size = 2
89 gp.type.s.0.member.0 = classjavalangObject
90 gp.type.s.0.member.1 = classjavautilStack
91
92 # FUNCTION FILES
93 #
```

```
94  gp.fs.0.size = 8
95  gp.fs.0.func.0 = eCrash.functionFiles.Stack_probabilities.FF00_Stack
96  gp.fs.0.func.0.nc = nc0
97  gp.fs.0.func.1 = eCrash.functionFiles.Stack_probabilities.FF01_pop
98  gp.fs.0.func.1.nc = nc1
99  gp.fs.0.func.2 = eCrash.functionFiles.Stack_probabilities.FF02_pop
100 gp.fs.0.func.2.nc = nc2
101 gp.fs.0.func.3 = eCrash.functionFiles.Stack_probabilities.FF04_push
102 gp.fs.0.func.3.nc = nc4
103 gp.fs.0.func.4 = eCrash.functionFiles.Stack_probabilities.FF05_push
104 gp.fs.0.func.4.nc = nc5
105 gp.fs.0.func.5 = eCrash.functionFiles.Stack_probabilities.FF07_peek
106 gp.fs.0.func.5.nc = nc7
107 gp.fs.0.func.6 = eCrash.functionFiles.Stack_probabilities.FF11_Object
108 gp.fs.0.func.6.nc = nc11
109 
110 # NODE CONSTRAINTS
111 #
112 # size
113 gp.nc.size = 8
114 # Stack() > IMPLICITPARAMETER
115 gp.nc.0 = ec.gp.GPNodeConstraints
116 gp.nc.0.name = nc0
117 gp.nc.0.returns = classjavautilStack
118 gp.nc.0.size = 0
119 gp.nc.0.prob = 1.0
120 # Object Stack.pop() > RETURN
121 gp.nc.1 = ec.gp.GPNodeConstraints
122 gp.nc.1.name = nc1
123 gp.nc.1.returns = classjavalangObject
124 gp.nc.1.size = 1
125 gp.nc.1.child.0 = classjavautilStack
126 gp.nc.1.prob = 1.0
127 # Object Stack.pop() > IMPLICITPARAMETER
128 gp.nc.2 = ec.gp.GPNodeConstraints
129 gp.nc.2.name = nc2
130 gp.nc.2.returns = classjavautilStack
131 gp.nc.2.size = 1
132 gp.nc.2.child.0 = classjavautilStack
133 gp.nc.2.prob = 1.0
134 # public Object Stack.push(Object) > RETURN
135 gp.nc.3 = ec.gp.GPNodeConstraints
136 gp.nc.3.name = nc4
137 gp.nc.3.returns = classjavalangObject
138 gp.nc.3.size = 2
139 gp.nc.3.child.0 = classjavautilStack
140 gp.nc.3.child.1 = classjavalangObject-S
141 gp.nc.3.prob = 1.0
142 # Stack.push(Object) > IMPLICITPARAMETER
143 gp.nc.4 = ec.gp.GPNodeConstraints
144 gp.nc.4.name = nc5
145 gp.nc.4.returns = classjavautilStack
146 gp.nc.4.size = 2
147 gp.nc.4.child.0 = classjavautilStack
148 gp.nc.4.child.1 = classjavalangObject-S
149 gp.nc.4.prob = 1.0
150 # Object Stack.peek() > RETURN
151 gp.nc.5 = ec.gp.GPNodeConstraints
152 gp.nc.5.name = nc7
```

```
153 gp.nc.5.returns = classjavalangObject
154 gp.nc.5.size = 1
155 gp.nc.5.child.0 = classjavautilStack
156 gp.nc.5.prob = 1.0
157 # Object() > IMPLICITPARAMETER
158 gp.nc.6 = ec.gp.GPNodeConstraints
159 gp.nc.6.name = nc11
160 gp.nc.6.returns = classjavalangObject
161 gp.nc.6.size = 0
162 gp.nc.6.prob = 1.0
163
164
165 ### MUT SPECIFIC PARAMETERS ###
166
167 # public synchronized int  java.util.Stack.search(java.lang.Object)
168 gp.nc.7 = ec.gp.GPNodeConstraints
169 gp.nc.7.name = nc12
170 gp.nc.7.returns = TREE
171 gp.nc.7.size = 2
172 gp.nc.7.child.0 = class javautilStack
173 gp.nc.7.child.1 = classjavalangObject-S
174 gp.nc.7.prob = 1.0
175
176 gp.fs.0.func.7 = eCrash.functionFiles.Stack_probabilities.FF09_search
177 gp.fs.0.func.7.nc = nc12
```

Listing B.1: Example ECJ Parameter File for `Stack`'s `search` method.

```
1  package eCrash.functionFiles.Stack;
2  import eCrash.MyGPNode;
3
4  public class FF09_search extends MyGPNode {
5          public FF09_search () {
6                  m.methodId = "FF09_search";
7                  m.methodSignature = "Stack.search(Object)";
8
9                  m.isConstant = false;
10                 m.isConstructor = false;
11                 m.modifiers = 33;
12
13                 m.methodName = "search";
14                 m.methodClass = "class java.util.Stack";
15
16                 m.methodReturnClass = "int";
17                 m.methodReturnIsPrimitive = true;
18                 m.methodReturnIsArray = false;
19
20                 m.methodParametersClass.add("java.lang.Object");
21                 m.methodParametersIsPrimitive.add(false);
22                 m.methodParametersIsArray.add(false);
23
24                 m.associatedItem = 3; // PARAMETER#0
25         }
26 }
```

Listing B.2: Example ECJ Function File for `Stack`'s `search` method.

Appendix C

Resumen en Español

Introducción

La Prueba de Software ("Software Testing") es el proceso de verificar una aplicación con el objetivo de detectar errores y de comprobar si se cumple con los requisitos especificados. Es un proceso costoso, que típicamente consume aproximadamente la mitad de los costos totales relacionados con el desarrollo de Software; automatizar el proceso de generación de Datos de Prueba es, pues, vital para avanzar el estado del arte en lo que respecta a las Pruebas de Software. El empleo de Algoritmos Evolutivos para generación de Datos de Prueba es, muchas veces, referido como "Evolutionary Testing". El objetivo del Evolutionary Testing es encontrar un conjunto de Programas de Prueba que cumplan con un criterio de prueba particular.

El objetivo de este trabajo es el desarrollo de una solución basada en Programación Genética para evolucionar Datos de Prueba para la Prueba Unitaria de programas Orientados a Objetos. La técnica propuesta para el Evolutionary Testing de Software Orientado a Objetos abarca la representación de Casos de Prueba utilizando Programación Genética Fuertemente Tipada ("Strongly-Typed Genetic Programming"). La evaluación de la calidad de los Casos de Prueba incluye la instrumentalización de los Objetos de Prueba, y también la ejecución de los Programas de Prueba generados con la intención de recoger información de "tracing" para obtener indicadores de cobertura. Se pretende así orientar, de manera eficiente, el proceso de búsqueda hacia la consecución de la plena cobertura estructural del programa bajo prueba.

Los objetivos principales de este trabajo fueron los de definir estrategias para encontrar respuestas a los desafíos presentados por el paradigma de Programación Orientada a Objetos en el contexto de la automatización de Pruebas de Software, y de proponer metodologías para mejorar la eficiencia y la eficacia de las técnicas de Evolutionary Testing.

Las Contribuciones más relevantes aportadas en este trabajo incluyen:

- la introducción de una nueva estrategia para la evaluación de Programas de Prueba y para guiar la búsqueda;
- la presentación de una metodología para reducir el Dominio de Entrada, basada en el concepto de Análisis de Pureza ("Purity Analysis").
- sugerir una metodología adaptativa para promover la introducción de instrucciones relevantes, a través de la Mutación, dentro de los Programas de Prueba generados; y
- la propuesta de una metodología de Reutilización de Objetos para metodologías de Evolutionary Testing basadas en Programación Genética, que permite que una sola instancia de un objeto sea utilizada como un parámetro de una función varias veces.

Los avances alcanzados resultaran en el desarrollo e implementación de la herramienta de generación de Datos de Prueba "eCrash", que incorpora las técnicas de Evolutionary Testing para el Software Orientado a Objetos propuestas.

Antecedentes y Motivación

La Prueba Unitaria ("Unit Testing") es una forma de probar el correcto funcionamiento de las unidades más pequeñas del Software – los Objetos de Prueba – probándolos en un ambiente aislado. La Prueba Unitaria es realizada ejecutando los Objetos de Prueba en diversos panoramas, usando Casos de Prueba ("Test Cases") relevantes e interesantes; un Conjunto de Casos de Prueba ("Test Set") se considera adecuado con respecto a un criterio dado si la totalidad de los Casos de Prueba en este sistema satisface este criterio. Una Prueba Unitaria para Software Orientado a Objetos consiste en una secuencia de llamadas a métodos, que define el escenario de la

prueba. Durante la ejecución del Caso de Prueba, todos los objetos que participan se crean y ponen en estados particulares con una serie de llamadas a métodos, y cada Caso de Prueba se centra en la ejecución de un método público particular – el Método Bajo Prueba ("Method Under Test").

La mayoría de la investigación en Pruebas de Software se ha hecho teniendo en mente el Software Orientado a Procedimientos; sin embargo, los métodos tradicionales no se pueden aplicar a Software Orientado a Objetos sin ser previamente adaptados. En Software Orientado a Objetos, las clases y los objetos se consideran típicamente las unidades más pequeñas que se pueden probar de forma aislada. Un objeto almacena su estado en campos, y expone su comportamiento a través de métodos. Ocultar el estado interno y requerir que toda la interacción se haga a través de los métodos de un objeto se conoce como Encapsulación de Datos – un principio fundamental de la programación Orientada a Objetos.

Los Algoritmos Evolutivos emplean evolución simulada como estrategia de búsqueda para evolucionar soluciones candidatas para un problema, usando operadores inspirados por la genética y la selección natural. La Programación Genética, en particular, es una especialización de los Algoritmos Genéticos generalmente asociada a la evolución de las estructuras con forma de árbol; se centra en la creación automática de programas de computadora por medio de la evolución, por lo que es especialmente adecuada para la representación y la evolución de Casos de Prueba.

Los nodos de un árbol en Programación Genética son, por lo general, no tipados – es decir, todas las funciones son capaces de aceptar todos los argumentos posibles. Sin embargo, técnicas de Programación Genética no tipadas no son adecuadas para representar Casos de Prueba para Software Orientado a Objetos; inversamente, la Programación Genética Fuertemente Tipada permite la definición de tipos para variables, constantes, argumentos, y valores devueltos. La única restricción es que el tipo de dato para cada elemento debe ser especificado de antemano en el Conjunto de Funciones ("Function Set"); esto hace que el proceso de inicialización y todas las operaciones genéticas sólo construyan árboles sintácticamente correctos.

El Evolutionary Testing consiste en la aplicación de Algoritmos Evolutivos a la generación de Datos de Prueba. El objetivo del Evolutionary Testing es encontrar un conjunto de Casos de Prueba que satisfaga un determinado criterio – como la cobertura estructural total de los Objetos de Prueba. El objetivo de la prueba debe ser definido numéricamente, y funciones de aptitud ("fitness") adecuadas, que ofrezcan orientación a la búsqueda y evalúen la calidad de las soluciones candidatas, deben ser

definidas. El espacio de búsqueda es el conjunto de entradas posibles al Objeto de Prueba; en el caso particular de los programas Orientados a Objetos, el dominio de entrada incluye los parámetros de los métodos públicos del Objeto de Prueba. Como tal, la meta de la búsqueda evolutiva es encontrar Casos de Prueba que definan escenarios de estado interesantes para las variables que se pasarán, como argumentos, en la llamada al Método Bajo Prueba. Uno de los desafíos más apremiantes que enfrentan los investigadores es el Problema del Estado ("State Problem"), que ocurre porque los objetos almacenan información en campos que están protegidos contra la manipulación externa – y que únicamente se pueden acceder a través de los métodos públicos que exponen el estado interno de los objetos.

La definición de un Conjunto de Casos de Prueba que logre la cobertura estructural implica la generación de Casos de Prueba complejos e intrincados a fin de definir escenarios de estado elaborados, y requiere la definición de metodologías cuidadosamente afinadas que promuevan el recorrido de estructuras problemáticas.

Metodología

La metodología de Evolutionary Testing para Software Orientado a Objetos propuesta implica la codificación de soluciones potenciales (i.e., Casos de Prueba) como individuos de Programación Genética Fuertemente Tipada. La Programación Genética Fuertemente Tipada es especialmente adecuada para representar y desarrollar programas Orientados a Objetos, que pueden ser representados como Árboles de Llamadas a Métodos ("Method Call Trees"). Un Árbol de Llamadas a Métodos es compuesto por nodos-método: cada una de ellos representa un método que más tarde será incluido en el Caso de Prueba decodificado, y el nodo raíz representa el Método Bajo Prueba.

En la Programación Genética Fuertemente Tipada, los tipos se definen *a priori* en el Conjunto de Funciones y definen las restricciones implicadas en la construcción de los Árboles de Llamadas a Métodos; es decir, el Conjunto de Funciones contiene el conjunto de instrucciones que el algoritmo puede utilizar cuando está construyendo Casos de Prueba. Esta característica permite que el proceso de inicialización y las distintas operaciones genéticas sólo construyan Árboles de Llamadas a Métodos sintácticamente correctos, restringiendo así el espacio de búsqueda al conjunto de Programas de Prueba compilables. El Conjunto de Funciones se define de forma totalmente au-

tomática, sólo en base a la información contenida en el "Test Cluster" (i.e., el conjunto transitivo de clases que son pertinentes para probar la Clase Bajo Prueba).

Para que un Caso de Prueba sea ejecutado, el genotipo (i.e., el Árbol de Llamadas a Métodos) debe ser decodificado en el fenotipo (i.e., el Caso de Prueba); esto puede lograrse covirtiendo el árbol en una lista, a través de un algoritmo de búsqueda en profundidad. La calidad de un determinado Caso de Prueba se relaciona con los nodos del Grafo de Control de Flujo (que representa el Método Bajo Prueba) que son los objetivos de la búsqueda evolutiva en una determinada fase del proceso; Casos de Prueba que ejerciten estructuras poco (o nada) exploradas son favorecidos, con el objetivo de alcanzar la meta principal del proceso de generación de Datos de Prueba – encontrar un conjunto de Casos de Prueba que logre la plena cobertura estructural del Objeto de Prueba.

La herramienta eCrash 'encarna' la metodología de Evolutionary Testing para Software Orientado a Objetos presentada en esta Tesis. Es un prototipo de herramienta de generación de datos de prueba basado en Java, y se desarrolló durante el curso del doctorado para apoyar la integración de los pasos de la investigación. El logro del más alto nivel de automatización posible ha sido siempre una de las preocupaciones principales subyacentes en su desarrollo e implementación; la falta de automatización es uno de los principales problemas que enfrentan los Probadores de Software hoy en día, y un importante obstáculo que todavía impide que el Software Orientado a Objetos sea adecuadamente probado y validado.

Contribuciones para la Mejora de las Metodologías de Evolutionary Testing para Software Orientado a Objectos basadas en Programación Genética

Contribuciones significativas para mejorar la aplicabilidad y el rendimiento de metodologías de Evolutionary Testing se han logrado como resultado de esta investigación. Estos avances serán brevemente descritos en las secciones siguientes.

Una estrategia para la evaluación de Programas de Prueba para el Evolutionary Testing de Software Orientado a Objetos

La generación de Datos de Prueba es un tema de investigación difícil, especialmente si el objetivo es implementar una solución automatizada, adaptable y de bajo costo. El Problema del Estado ("State Problem") de los programas Orientados a Objetos, en particular, requiere la definición de metodologías que promuevan la cobertura de estructuras problemáticas y de difíciles caminos de control de flujo. Nos propusimos hacer frente a este desafío a través de una nueva estrategia para la evaluación de Programas de Prueba y para la orientación de la búsqueda. La técnica propuesta implica la definición de nodos ponderados en el Grafo del Control de Flujo, y permite que Casos de Prueba inviables (i.e., aquellos que terminan prematuramente debido a una "Runtime Exception") sean considerados en ciertas etapas de la búsqueda evolutiva – a saber, una vez que los Casos de Prueba viables que están siendo generados dejen de ser interesantes, porque solo ejercitan estructuras recurrentemente atravesadas. En conjunto, con el impacto de los operadores evolutivos de Mutación y Recombinación, se puede lograr un buen compromiso entre la intensificación y la diversificación de la búsqueda.

El empleo de Análisis de Pureza para la reducción del dominio de entrada de Software Orientado a Objetos

La metodología de reducción del dominio de entrada para la eliminación de variables irrelevantes de problemas de Evolutionary Testing propuesto se basa en el concepto de la Análisis de Pureza, y proporciona un medio para identificar y eliminar automáticamente las entradas del Conjunto de Funciones que no contribuyen en la definición de escenarios de prueba interesantes. Este proceso también asegura que los Programas de Pruebas no sean inviables debido a la inclusión de instrucciones sin interés; esto es particularmente importante en el contexto de nuestra metodología, dado que la estrategia de evaluación de Casos de Prueba definida considera programas inviables en ciertas etapas de la búsqueda. Las observaciones realizadas indican que la estrategia de reducción del dominio de entrada que se presenta tiene un efecto altamente positivo en la eficiencia del algoritmo de generación de Casos de Prueba: se gasta menos tiempo de cómputo para lograr resultados.

Una metodología adaptativa para el Evolutionary Testing de Software Orientado a Objetos

La inclusión de instrucciones pertinentes en los Programas de Prueba generados (por medio del operador de Mutación) es también la justificación para la metodología adaptativa definida para actualizar dinámicamente las probabilidades de selección de las entradas definidas en el Conjunto de Funciones. Esta estrategia obtiene información de las soluciones anteriormente producidas y evaluadas, con el fin de promover la diversidad y viabilidad de los Programas de Prueba. Los estudios experimentales indican una mejora considerable en la eficiencia del algoritmo en comparación con su versión estática, al tiempo que introduce una sobrecarga insignificante.

Permitir la reutilización de objetos en metodologías de Evolutionary Testing para Software Orientado a Objetos basadas en Programación Genética

La estrategia para la reutilización de objetos propuesta permite la generación de Casos de Prueba que ejercitan estructuras específicas de Software que no se pudrían ejercitar de otra manera, por lo que es especialmente importante si criterios de adecuación estructurales ("White-Box") son empleados. Reutilización de Objetos significa que un objeto puede ser pasado a varios métodos, o varias veces al mismo método, como argumento. La metodología propuesta para reutilización de objetos implica la definición de un nuevo tipo de nodos de Programación Genética – los "At-Nodes" – que 'apuntan' a otros nodos, con lo que permiten la reutilización de las referencias a objetos devueltas por las funciones correspondientes a los sub-árboles apuntados. Los "At-Nodes" pueden ser eliminados de un árbol; esta característica particular permite evitar la reformulación de otros operadores evolutivos, como los de Mutación y Recombinación. Además de mejorar la eficacia de la búsqueda, los estudios experimentales realizados muestran que la metodología propuesta es capaz de aumentar el rendimiento del proceso de generación de Datos de Prueba: las soluciones producidas tienen menor tamaño y menor complejidad estructural, y el promedio de viabilidad de los Programas de Prueba generados es mejorado.

Conclusiones y Trabajo Futuro

Los objetivos iniciales de este trabajo eran los de contribuir positivamente para mejorar los niveles de automatización y rendimiento del (a menudo descuidado, pero enormemente importante) proceso de Pruebas de Software, y de investigar la pertinencia de aplicar Algoritmos Evolutivos a los problemas de generación de Datos de Prueba. Creemos que es posible afirmar que este objetivo se logró, no sólo como resultado de las propuestas formuladas para mejorar la eficiencia y la eficacia de metodologías de Evolutionary Testing para Software Orientado a Objetos, sino también porque esta investigación resultó en el desarrollo de una herramienta automatizada que demuestra la aplicabilidad de las metodologías propuestas.

La herramienta eCrash incorpora los avances en el área de Evolutionary Testing propuestos en esta Tesis. A pesar de ser todavía un prototipo, eCrash es totalmente funcional y es aplicable a una amplia gama de Objetos de Prueba. No obstante, estamos trabajando activamente en el desarrollo de una aplicación estable, fácil de usar y bien documentada; los planes para el futuro próximo incluyen la publicación de una versión de eCrash integrada en el IDE Eclipse, que pueda ser utilizada por los Probadores de Software en un entorno de producción, y por los investigadores de Evolutionary Testing por igual.

También tenemos planes para hacer frente a algunas cuestiones de investigación que todavía no hemos tenido la oportunidad de estudiar, a saber: posibilitar la prueba de los métodos no-públicos (a través de la interfaz pública de un objeto); explorar el tema de “Search Space Sampling”, que se ocupa de la inclusión de todas las variables relevantes para un determinado Objeto de Prueba en el problema de generación de Datos de Prueba; abordar el problema de generación de ’oráculos’, e investigar la posibilidad de automatizar un mecanismo para comprobar que las salidas de un programa son correctass y de acuerdo con los entradas; y, finalmente, experimentar con sistemas paralelos a fin de aumentar el rendimiento de nuestra metodología.