Information and Software Technology 51 (2009) 1534–1548

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Test Case Evaluation and Input Domain Reduction strategies for the Evolutionary Testing of Object-Oriented software

José Carlos Bregieiro Ribeiro [a,*], Mário Alberto Zenha-Rela [b], Francisco Fernández de Vega [c]

[a] Polytechnic Institute of Leiria, Morro do Lena, Alto do Vieiro, Leiria, Portugal
[b] University of Coimbra, CISUC, DEI, 3030-290 Coimbra, Portugal
[c] University of Extremadura, C/Sta Teresa de Jornet, 38 Mérida, Spain

ARTICLE INFO

Article history:
Available online 5 July 2009

Keywords:
Evolutionary Testing
Search-Based Software Engineering
Test Case Evaluation
Input Domain Reduction

ABSTRACT

In Evolutionary Testing, meta-heuristic search techniques are used for generating test data. The focus of our research is on employing evolutionary algorithms for the structural unit-testing of Object-Oriented programs. Relevant contributions include the introduction of novel methodologies for automation, search guidance and Input Domain Reduction; the strategies proposed were empirically evaluated with encouraging results.

Test cases are evolved using the Strongly-Typed Genetic Programming technique. Test data quality evaluation includes instrumenting the test object, executing it with the generated test cases, and tracing the structures traversed in order to derive coverage metrics. The methodology for efficiently guiding the search process towards achieving full structural coverage involves favouring test cases that exercise problematic structures. Purity Analysis is employed as a systematic strategy for reducing the search space.

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. Yet, it remains the primary method through which confidence in software is achieved [6]. A large amount of the resources spent on testing are applied on the difficult and time consuming task of locating quality test data; automating this process is vital to advance the state-of-the-art in software testing. However, automation in this area has been quite limited, 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 [25].

Meta-heuristic 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 [4] – 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 [10].

The application of Evolutionary Algorithms to test data generation is often referred to as *Evolutionary Testing* [39] or *Search-Based Testing* [25]. Approaches have been proposed that focus on the usage of Genetic Algorithms [16,17,39,48], Ant Colony Optimization [22], Genetic Programming [38], Strongly-Typed Genetic Programming [43,45], and Memetic Algorithms [1].

Evolutionary Testing is an emerging methodology for automatically generating high quality test data. 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 this technique to the automatic generation of unit-test cases for procedural software [24,25]. The application of search-based strategies for Object-Oriented unit-testing is, however, fairly recent [39] and is yet to be investigated comprehensively [11].

The focus of our research is precisely on developing a solution for employing Evolutionary Algorithms for generating test sets for the structural unit-testing of Object-Oriented programs. Our approach involves representing and evolving test cases using the Strongly-Typed Genetic Programming technique [28]. 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 with which to derive coverage metrics. The aim is that of efficiently guiding the search process towards achieving full structural

* Corresponding author. Tel.: +351 965522037.
E-mail addresses: jose.ribeiro@estg.ipleiria.pt, jcbribeiro@gmail.com (J.C.B Ribeiro), mzrela@dei.uc.pt (M.A. Zenha-Rela), fcofdez@unex.es (F. Fernández de Vega).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.06.009

coverage of the program under test. These concepts have been implemented into the *eCrash* automated test case generation tool – which will be described below.

Our main goals are those of defining strategies for addressing the challenges posed by the Object-Oriented paradigm and of proposing methodologies for enhancing the efficiency of search-based testing approaches. The primary contributions of this work are the following:

- Presenting a strategy for Test Case Evaluation and search guidance, which 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.
- Introducing a novel 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 test scenarios.

Additionally, our methodology for automated test case generation is thoroughly described and validated through a series of empirical studies performed on standard Java classes.

This article is organized as follows. In the next Section, we start by introducing the concepts underlying our research. Next, related work is reviewed and contextualized. In Section 4, our test case generation methodology and the *eCrash* tool are described. The experiments conducted in order to validate and observe the impact of our proposals are discussed in Section 5, with special emphasis being put on studying the novel Test Case Evaluation and Input Domain Reduction strategies. The concluding Section presents some final considerations, the most relevant contributions, and topics for future work.

2. Background and terminology

In Evolutionary Testing, meta-heuristic search techniques are employed to select or generate test data; this section presents the most important Software Testing and Evolutionary Algorithms aspects related with this interdisciplinary area. Special attention is paid to the concepts of particular interest to our technical approach.

2.1. Software testing

Software testing is the process of exercising an application to detect errors and to verify that it satisfies the specified requirements [21]. 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 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.

Distinct levels of testing include functional (black-box) and structural (white-box) testing [6]. Traditional *structural adequacy criteria* include branch, data-flow and statement coverage; 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. The metrics for measuring the thoroughness of a test set can be extracted from the structure of the target object's source code, or even from compiled code (e.g., Java bytecode).

The evaluation of the quality of a given test set and the guidance to the test case selection using structural criteria generally requires the definition of an underlying model for program representation – usually a *Control-Flow Graph* (e.g., Fig. 4). The Control-Flow Graph is an abstract representation of a given method in a class; control-flow testing criteria can be derived based on such a program representation to provide a theoretical and systematic mechanism to assess the quality of the test set [29]. Two well known *control-flow testing standards* to derive testing requirements from the Control-Flow Graph are the all-nodes and all-edges criteria [42].

The observations needed to assemble the metrics required for the evaluation of test data suitability can be collected by abstracting and modelling the behaviours programs exhibit during execution, either by static or dynamic analysis techniques [40]. Static analysis involves the construction and analysis of an abstract mathematical model of the system (e.g., symbolic execution); testing is performed without executing the method being tested, but rather this abstract model. This type of analysis is complex, and often incomplete due to the simplifications in the model. In contrast, *dynamic analysis* involves executing the actual test object and monitoring its behaviour; while it may not be possible to draw general conclusions from dynamic analysis, it provides evidence of the successful operation of the software.

Dynamic monitoring of structural entities can be achieved by *instrumenting* the test object, and *tracing* the execution of the structural entities traversed during test case execution. Instrumentation is performed by inserting probes in the test object.

2.1.1. Object-Oriented Software Testing

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

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 [5].

A unit-test case for Object-Oriented software consists of a *Method Call Sequence*, which defines the test scenario. During test case 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 *Method Under Test*. It is not possible to test the operations of a class in isolation; testing a single class involves other classes, i.e., classes that appear as parameter types in the method signatures of the *Class Under Test*. The transitive set of classes which are relevant for testing a particular class is called the *test cluster* [43].

In summary, the process of performing unit-testing on Object-Oriented programs usually requires [44]:

- at least, an instance of the Class Under Test;
- additional objects, which are required (as parameters) for the instantiation of the Class Under Test and for the invocation of the Method Under Test – 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.

2.2. Evolutionary Algorithms

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. The best