Flycatcher

Automatic unit test generation for JavaScript

Jerome de Lafargue
jdd08@ic.ac.uk

MEng Computing Individual Project Interim Report

Supervisor: Susan Eisenbach
Second Marker: John Smith
Technical Supervisor: Tristan Allwood

Department of Computing
Imperial College London

Contents

Chapter 1

Introduction

Notes on [6]

- testing major part of the development process, crucial for quality and accounts for 50% or more of the development effort
- manual process is tedious, costly and difficult and the testing is often biased
- exhaustive enumeration of inputs is infeasible for a reasonable program, coverage gives a good indication of test quality but full coverage is often infeasible too therefore ideally ATDG should achieve the best possible coverage
- random methods are easy to implement but unreliable and unlikely to discover deep errors
- size and complexity of real software means a lot of the research deals with toy examples/language subsets. ATDG 'undecidable' problem.
- metaheuristic search techniques use an *objective function* that estimates the value of a solution
- the execution tree of most programs is infinite [3]

1.1 Motivation

1.2 Aim

Chapter 2

Background

In this section we will start by giving an overview of software testing, with particular emphasis on aspects of it that are relevant to this project. We will then take a look at the state of the art in automatic test generation in order to understand the approach that will be used for Flycatcher. Finally we will explain why we chose JavaScript as our target language and describe features of it that are of interest to us for this project.

2.1 Dynamic testing

2.1.1 Overview

We can define the activity of dynamic testing as testing that requires execution of the software with test data as input [4] and characterise it with respect to three parameters namely the point of view taken by the tester, the target of the tests and the stage of development at which they are executed. The point of view taken concerns the knowledge of the software under test and can be divided into three categories, structural (white-box testing) testing, functional (black-box testing) and a hybrid of the two (grey-box testing). The target of the tests refers to their granularity, from testing specific units of code (unit testing) to an entire integrated system (system testing). The stage at which the tests are undertaken determines whether they are regression tests, alpha-tests, beta-tests, acceptance tests *etc.* With Flycatcher we hope to generate suites of structural tests, focused at the unit level of object-oriented classes, most likely to perform incremental regression testing. Hence, structural testing (our point of view), unit testing (our target level) and regression testing (our development stage) will be described in more detail in this section.

2.1.2 Structural testing

The goal of structural testing is to test the internal workings [6] of an application in order to verify that it does not contain errors. While functional testing determines whether the software provides the required functionality, structural testing ensures that it will not crash under any circumstances, regardless of how it is called. It concerns *how* the software operates, rather than *what* it can do. As a result, the measure to determine good structural testing is code coverage — the amount of code that we cover during the testing process. In practice, guaranteeing that all possible executions of a piece of software are error-free is an undecidable problem, but coverage gives us a good estimate of the quality of our tests.

Test coverage

Test oracles

There are many different ways to gage the amount of software covered during the testing process. Predicate or branch coverage evaluates the percentage of control flow branches covered where a branch exists wherever the software has the possibility to go down a different path such as upon `if` statements, loops *etc.*

2.1.3 Unit testing

2.1.4 Regression testing

2.2 Automatic test case generation

In this project, we are concerned with generating structural unit tests, as defined earlier, for an object-oriented, dynamic language. At the core of any automatic test case generation task is the generation of input data for the tests, which we will refer to from now on as ATDG (Automatic Test Data Generation). For object-oriented tests other considerations are the construction of objects and the ordering of statements in test cases. However, first and foremost we will look at techniques for generating the input data itself, as would be done for tests in a purely procedural language. Then we will look at methods for generating object-oriented tests and the specific challenges posed by dynamic languages for ATDG.

2.2.1 Overview

Edvardssvon's classification [1] is based on the idea that there are three approaches to automatic test data generation: *random*, (randomly explore the search space) *goal-oriented* (work towards specific goals)

2.2.2 Static approach

Static structural test data generation is based on information available from the static analysis of the program, without requiring that the program is actually executed [6]. Static program analysis produces control flow information that can be used to generate execution paths that can be used for static test data generation. Every time control flow branches, e.g. at `if` statements, there is a corresponding predicate or branch condition. These predicates can be collected along a path and conjoined to form the path predicate. By solving the path predicate in terms of the input variables, we can obtain test data that executes that path. However, in order to rewrite the path predicate in terms of the input variables we need to take into account the execution of the program. This is known

2.2.3 Dynamic approach

Random ATDG

Search-based ATDG

2.2.4 Hybrid approach

2.2.5 Object-oriented test case generation

2.2.6 Challenges of dynamic languages

2.3 JavaScript

2.3.1 Prominence

2.3.2 Idiosyncratic features

2.3.3 Object-oriented programming

This is DART [2]. This is RuTeG [5].

Bibliography

[1] Edvardsson, J. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering* (1999), no. x, pp. 21–28.

[2] Godefroid, P., Klarlund, N., and Sen, K. Dart: directed automated random testing. In *ACM Sigplan Notices* (2005), vol. 40, ACM, pp. 213–223.

[3] King, J. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[4] Mahmood, S. A systematic review of automated test data generation techniques. *School of Engineering, Blekinge Institute of Technology Box 520* (2007).

[5] Mairhofer, S. Search-based software testing and complex test data generation in a dynamic programming language. *Master's thesis, Blekinge Institute of Technology* (2008).

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