Flycatcher

Automatic unit test generation for JavaScript

Jerome de Lafargue

jdd08@ic.ac.uk

MEng Computing Individual Project Interim Report

Supervisor: Susan Eisenbach
Technical Supervisor: Tristan Allwood

Department of Computing
Imperial College London

Contents

Chapter 1

Introduction

Notes on [9]

- 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 [7]
- as more and more people rely on software it will be required to be of greater quality

1.1 Motivation

1.2 Aim

Chapter 2

Background

In this chapter 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 data generation (ATDG) in order to understand the approach that will be used for Flycatcher. Because the tests will be object-oriented, method call sequences need to be generated and we will look at the state of the art for doing that too. 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 software 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 [8] and characterise it with respect to three parameters namely the amount of knowledge assumed by the tester, the target of the tests and the stage of development at which they are executed. The amount of knowledge of the software under test 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, unit testing and regression testing 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 [9] 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 tries to ensure that it does not crash under any circumstances, regardless of how it is called. It concerns *how* well the software operates, its structure, rather than *what* it can do, its function. As a result, the measure to determine good structural testing is the code covered during the testing process — code coverage. It gives us an idea of the amount of code that should be bug free. However, there are various types of code coverage criteria and the confidence that our code is bug free varies depending on which one is chosen.

Test coverage

Edvardsson lists the most cited criteria [2], from weakest to strongest:

- **Statement Coverage** Each statement must be executed at least once.
- **Branch[1]/Decision Coverage** Each branch condition must evaluate to true and false.
- **Condition/Predicate Coverage** Each clause within each branch condition must evaluate to true and false.
- **Multiple-condition Coverage** Each combination of truth values of each clause of each condition must be executed.
- **Path[1] Coverage** Each path in the control flow graph[1] must be traversed.

  The stronger criteria of condition, multiple-condition and path coverage are often infeasible to achieve for programs of more than moderate complexity, and thus branch coverage has been recognised as a basic measure for testing [2].

2.1.3 Unit testing

Unit testing consists in testing individual and independently testable units of source code [11]. Therefore, unit testing is made easier if the code is designed in a modular way. The nature of the units depends on the programming language and environment but is often a class or a function. As opposed to system tests which can be aimed at the client, unit tests are usually white-box tests. Although they do not guarantee that the overall software works

[1]For an explanation of program analysis terminology such as branch, path and control flow graph see Appendix A.

as required, they give confidence in specific units of code and narrow down errors, helping the development process. In Flycatcher, the target unit will be a JavaScript prototype, which will be introduced later in this chapter.

2.1.4 Regression testing

Automatically generating structural unit tests can be of great use for regression testing. Regression testing aims to ensure that enhancing a piece of software does not introduce new faults [11]. The difficulty in testing this is that programmers do not always appreciate the extent of their changes. Hence, having a suite of unit tests with good structural coverage can reduce this problem by verifying the software in a systematic, unbiased way.

2.2 Automatic test data generation

2.2.1 Overview

As can be seen in Mahmood's systematic review of ATDG techniques [8], many classifications exist for ATDG techniques. For our purposes, the first distinction that we need to make is between white-box, black-box and grey-box ATDG techniques, as for Flycatcher we are only interested in white-box testing. In the literature we found that white-box ATDG techniques are usually classified in two ways [8, 2, 16]. The first distinguishes whether the data is generated randomly, to cover a specific statement or to cover a specific path[2] — respectively *random*, *goal-oriented* and *path-oriented* ATDG [2]. The other classification of white-box ATDG focuses on the type of implementation: *static*, *dynamic* or a *hybrid* of the two [5, 9]. We will focus on the latter classification of structural testing as it is crucial to our choice of implementation for Flycatcher. Moreover, the former concerns the path selection stage of ATDG — whether data should be generated for a specific (path-oriented), unspecific (goal-oriented) or random path and this step is ignored in many recent ATDG techniques [16]. Figure 123 summarises what we believe is the most intuitive characterisation of ATDG techniques to date and the one which will guide our choice of implementation for Flycatcher. Many techniques can be found under each of the static, dynamic and hybrid implementation categories and we will only list the most noteworthy to us.

The choice of implementation for Flycatcher is dynamic random ATDG for our benchmark and dynamic search-based ATDG using genetic algorithms for our solution. The rest of the section will present in further detail the structural ATDG implementation categories we have chosen and the difficulties of ATDG for a dynamic language, so that we can understand the rationale behind this choice.

[2]this involves a path selection step, where paths are successively selected from the control flow graph to yield the best coverage for the chosen coverage criterion

2.2.2 Static ATDG

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 [9]. Static program analysis produces control flow information that can be used to select execution paths in order to try and achieve good coverage. The goal of ATDG is then to generate data that executes these paths.

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. Hence, for generation of test data statically a technique called symbolic execution [7] is used.

Symbolic execution gathers constraints along a simulated execution of a program path, where symbolic variables are used instead of actual values, such that the final path predicate can be rewritten in terms of the input variables. Solving the resulting system of constraints then yields the data necessary for the traversal of the given path [6, 7]. There are a lot of technical difficulties associated with symbolic execution [2, 10, 9]:

- the presence of input variable dependent loops can lead to infinite execution trees as they can be executed any number of times
- array references become problematic if the indexes are not constants but variables, as is typically the case
- features such as pointers and dynamically-allocated objects that rely on execution are hard to analyse statically
- static analysis is not possible for function calls to precompiled modules or libraries
- if the path constraint is non-linear, solving it becomes an undecidable problem
- even if the path constraint is linear, solving it can lead to very high complexity

Although various static solutions have been proposed for these issues [14, 4, 12], they often dramatically increase the complexity of the ATDG process. As a result, tools purely based on symbolic execution can typically handle only subsets of programming languages and are not applicable in industry. A better trend that has developed in the past decade, is the combination of concrete and symbolic execution, which tackles most of the aforementioned issues [13] — we will cover this type of ATDG implementation in the subsection on hybrid ATDG. Due to the numerous problems posed by purely static ATDG, its weakness with dynamic types and constructs [2, 16] and the complexity of building a fully-fledged symbolic executor for a language, we chose not to use static ATDG for the implementation of Flycatcher.

2.2.3 Dynamic ATDG

Random approach

Search-based approach

2.2.4 Hybrid ATDG

DART [3]. CUTE [15]. PEX [17]. EXE [1].

Although the hybrid approach to ATDG seems to offer the best of both worlds, dynamic ATDG is more suited to dynamic languages.

2.2.5 Challenges of dynamic languages

2.3 Object-oriented test case generation

2.4 JavaScript

2.4.1 Prominence

2.4.2 Idiosyncratic features

2.4.3 Object-oriented programming

Bibliography

[1] Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., and Engler, D. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC) 12*, 2 (2008), 10.

[2] 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.

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

[4] Goldberg, A., Wang, T., and Zimmerman, D. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis* (1994), ACM, pp. 80–94.

[5] Han, S., and Kwon, Y. An empirical evaluation of test data generation techniques. *Journal of Computing Science and Engineering 2*, 3 (2008), 274–300.

[6] King, J. A new approach to program testing. *Programming Methodology* (1975), 278–290.

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

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

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

[10] Meudec, C. Atgen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability 11*, 2 (2001), 81–96.

[11] Myers, G., Sandler, C., and Badgett, T. *The art of software testing*. Wiley, 2011.

[12] Offutt, A., Jin, Z., and Pan, J. The dynamic domain reduction procedure for test data generation. *Software-Practice and Experience 29*, 2 (1999), 167–194.

[13] Păsăreanu, C., and Visser, W. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer (STTT) 11*, 4 (2009), 339–353.

[14] Ramamoorthy, C., Ho, S., and Chen, W. On the automated generation of program test data. *Software Engineering, IEEE Transactions on*, 4 (1976), 293–300.

[15] Sen, K., Marinov, D., and Agha, G. *CUTE: A concolic unit testing engine for C*, vol. 30. ACM, 2005.

[16] Tahbildar, H., and Kalita, B. Automated software test data generation: Direction of research. *International Journal of Computer Science and Engineering 2*.

[17] Tillmann, N., and De Halleux, J. Pex–white box test generation for. net. *Tests and Proofs* (2008), 134–153.