Flycatcher

Automatic unit test generation for JavaScript

Interim Report

Author: Jerome de Lafargue
Supervisor: Susan Eisenbach
Technical Supervisor: Tristan Allwood

Department of Computing
Imperial College London

Contents

Chapter 1

Introduction

Software testing is a cornerstone of software engineering — one of the most common and effective ways to verify software quality and an effort that accounts for at least 50% of software development time [32]. With the fast-paced growth of the software industry, comes the need to test larger and more complex software on an unprecedented scale. Moreover, as software becomes increasingly ubiquitous, it is held to the highest standards of reliability and correctness, which further justifies testing it in a rigorous and exhaustive manner.

As a result, many attempts have been made to automate the testing effort, so that programs can be systematically and seamlessly tested, without requiring laborious, costly and error-prone manual input. The consequences of automated testing are very appealing: it reduces software maintenance and development costs, while increasing the robustness and ultimate quality of the software. Despite the fact that this area of research has taken time to develop, due to the intrinsic complexities of automatic test generation, it has now seemingly reached a stage where it can start to make a meaningful impact on software testing practice.

Decades of research have been devoted to automatic test generation for static languages and a multitude of tools have been developed. As the research area matures, it is arriving to a point where its techniques are no longer simply applicable to restricted programming language subsets or limited programs. Indeed, companies such as Microsoft employ automatic test generation tools on a regular basis to verify their software [26]. Yet, until very recently, dynamic programming languages had been left out of the equation — but their increasing popularity and a renewed interest in them prompts the need to start including them in the automatic testing research effort.

One such programming language that has been growing in popularity in the past few years is JavaScript, with new frameworks and libraries being released for it frequently. Software libraries that have gained wide acceptance, like `Node.js`[1] which supports the writing of highly-scalable internet applications, seem to confirm JavaScript's transition from a purely client-side browser language to an all-purpose one — at least for some. In recent studies [17], JavaScript appears amongst the most used programming languages in the world today. In other words, it seems that JavaScript is here to stay, at least for some time, and it makes sense to devote time to it.

Various test generation approaches exist: from straightforward but limited random generation to elaborate systems that combine static and dynamic analysis to provide strong software verification. Since much of the literature on automatic test generation focuses on static procedural languages and numerical input data types, many of the techniques found are not feasible or applicable to automatic test generation for JavaScript. However, some are more appropriate than others for our objective — to generate structural unit test suites for JavaScript objects — and it is those methods that we will explore during the course of this project.

This project is challenging for several reasons. First of all, it requires automatic test data generation, which is a challenging task in itself. Taking a random approach to the data generation problem is often not satisfactory, as the coverage achieved tends to be fairly poor. However, both the static and dynamic solutions that aim for a more systematic testing face major hurdles. For the static approach it is mainly to do with solving the constraints responsible for generating the test data, as this problem can become undecidable under certain circumstances. The dynamic approach depends on the execution of the program under test, and the number of executions needed for sufficient coverage can become infeasible. On top of these challenges that are common to most automatic test case generation initiatives, Flycatcher raises additional difficulties due to the fact that it targets not only an object-oriented language but a dynamic one. For instance, our testing tool will need to tackle issues such as method call sequence generation and dynamically-typed object instance generation, the latter rendered difficult by the absence of static types.

By investigating and bringing together the most relevant techniques, we believe that we can overcome the challenges encountered and build a tool for the structural unit-testing of JavaScript programs. The major contribution that this project hopes to make to the field is to extend the limited amount of work done regarding dynamic languages by proposing a tool for automatic unit test generation for, possibly restricted, JavaScript programs. As well as offering a potential tool to aid JavaScript developers, we hope that our work will be able to offer new insights into automatic test generation for a dynamic, object-oriented language and benefit future research in that direction.

[1] http://nodejs.org/

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 also need to be generated for the test cases 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 [18] 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 [20] 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 [7], 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 [7].

2.1.3 Unit testing

Unit testing consists in testing individual and independently testable units of source code [24]. 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 [24]. 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 [18], many classifications exist for ATDG techniques. For our purposes, the first distinction that we need to make is between white-box, black-box [28] 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 [18, 7, 32].

The first concerns the target selection stage of ATDG techniques: where either paths or individual nodes that contribute to the overall coverage criterion are successively selected from the control flow graph, so that test data that respectively traverses the path or reaches the node can be generated. When specific paths are targeted, the ATDG technique is known as *path-oriented* [7] whereas if a node is targeted then it is *goal-oriented*. When data is generated purely randomly *i.e.* there is no specific target, then as part of this classification the ATDG technique is simply *random*.

The other classification of white-box ATDG concerns the type of implementation: *static*, *dynamic* or a *hybrid* of the two [12, 20]. We will focus on the latter classification of structural testing as it governs our choice of implementation for Flycatcher. Moreover, the former concerns the path selection stage of ATDG and this step will be ignored in Flycatcher as it is in many recent ATDG techniques [32]. Figure 2.1 summarises what we believe is an intuitive characterisation of ATDG techniques with respect to this project and the one which will guide our choice of implementation. 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 this section will present in further detail

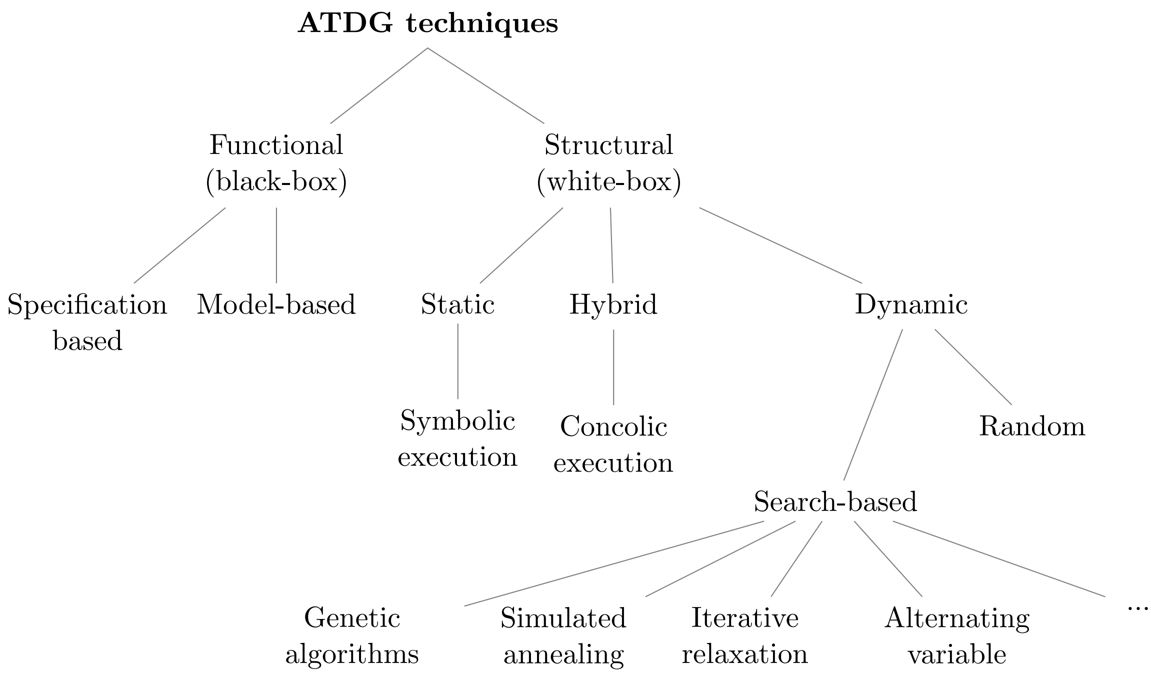

Figure 2.1: Overview of ATDG techniques

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.

2.2.2 Static test data generation

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 [20]. 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 [14] 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 [13, 14]. There are a lot of technical difficulties associated with symbolic execution [7, 21, 20]:

- the presence of input variable dependent loops can lead to infinite execution trees[2] as the loops 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 is 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 [29, 10, 25], 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 [26] — 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 [7, 32] and the complexity of building a fully-fledged symbolic executor for a language [7, 12], we chose not to use static ATDG for the implementation of Flycatcher.

2.2.3 Hybrid test data generation

The hybrid approach to ATDG consists in combining symbolic and concrete execution, which is known as *concolic execution* [26]. In other words, hybrid analysis tools run programs on actual inputs, while collecting symbolic constraints in order to direct the search for new inputs. In doing so, they avoid the main weaknesses of the static approach, such as solving non-linear constraints or dealing with dynamic structures. This type of technique has been popular in recent years, mainly because it overcame the limitations

[2] the execution paths followed during the symbolic execution of a procedure [14]

that prevented static ATDG techniques from being applied to industry software. Notable tools that implement it are DART [9], CUTE [31], JPF-SE [2], PEX [33], EXE [5] and KLEE [4].

Yet, although it deals with some limitations of static ATDG, hybrid ATDG still requires static analysis of the source code under test, which is unfeasible for Flycatcher, given the dynamically typed, object-oriented nature of JavaScript.

2.2.4 Dynamic test data generation

Dynamic structural test data generation is purely based on actual execution of the software. The program under test is run with, possibly randomly, selected input and feedback is collected at runtime regarding the chosen coverage objective [7]. The feedback is usually obtained through some form of instrumentation of the program that monitors the program flow. Inputs can be continually generated randomly, relying on probability to achieve the coverage objective — this is known as *random* test data generation and does not perform well in general [7]. On the other hand, inputs can be incrementally tuned based on the feedback (using different kinds of search methods) in order to satisfy the coverage objective — this is known as *search-based* test data generation [20], where the search-space is the control flow graph of a program. The main drawback of dynamic ATDG is that it is reliant on the speed of execution of a program and as the number of required executions to achieve satisfactory coverage may be high, this leads to an overall expensive process. Below we will present the random and search-based approaches in more detail, in order to understand which would be more suitable for Flycatcher.

Random approach

Random test data generation consists in producing inputs at random in the hope of achieving the chosen coverage criterion through probability. Although random test data generation is relatively simple to implement, it does not perform well in terms of coverage, as the chances of finding faults that are revealed by only a small percentage of program inputs are low [7]. In other words, it is difficult for it to exercise 'deep' features of a program that are exercised only through specific and unlikely paths. As a result, random ATDG only works well for straightforward programs. However, because it is the simplest ATDG technique and is considered to have the lowest acceptance rate [7], it is often used as a benchmark and is a suitable candidate for us to benchmark our Flycatcher application.

Search-based approach

Search-based test data generation uses heuristics to guide the generation of input data so that the inputs execute paths that contribute to the overall test coverage objective. This involves modelling the test coverage objective as a heuristic function or *objective function*, that evaluates the fitness of a chosen set of inputs with respect to a coverage objective. Based on those fitness values, many search techniques exist to find optimal inputs in order to achieve the desired coverage. Various objective functions exist and they are dependent on the ATDG method used. Some of the well-known search-based ATDG techniques are *alternating variable* (local search optimisation) [15, 8], *simulated annealing* [36, 35], *iterative relaxation* [11] and *genetic algorithms* [22, 23].

In 2008, Han and Kwon [12] conducted a robust empirical evaluation of search-based test data generation techniques, comparing iterative-relaxation, local search optimisation and genetic algorithms. Genetic algorithms came out on top of the other techniques regarding the rate of coverage and generality, both essential characteristics when considering ATDG techniques. However, genetic algorithms proved expensive both in time and resources, but this can be improved upon [12]. Although simulated annealing is not part of this empirical evaluation, we can see from [22] that although it performs as well as genetic algorithms, it appears to be less efficient.

Additionally, genetic algorithms offer another advantage: as opposed to most other ATDG techniques, they do not require static analysis of the program under test [12]. This advantage is especially significant in our case, as to the best of our knowledge, tools that produce control and data flow analysis information for JavaScript source code are not available. However, this is not surprising since the dynamic, object-oriented nature of JavaScript makes this task very laborious.

Consequently, due to their effectiveness in terms of coverage and the advantages they possess with regard to dynamic languages, genetic algorithms appear to be the most suitable for our implementation of ATDG for Flycatcher. We will therefore go on to explain what genetic algorithms are and how they can be applied to test data generation.

2.2.5 Genetic algorithms

Overview

Genetic algorithms attempt to model the robustness and flexibility of natural selection in order to guide a search[23]. They start with a randomly initialised *population* of potential candidate solutions, called *individuals* or *chromosomes*. The population is iteratively recombined and mutated to evolve successive populations, known as *generations*. The recombination takes the ‘fittest’ parent solutions and ‘breeds’ them to produce new off-

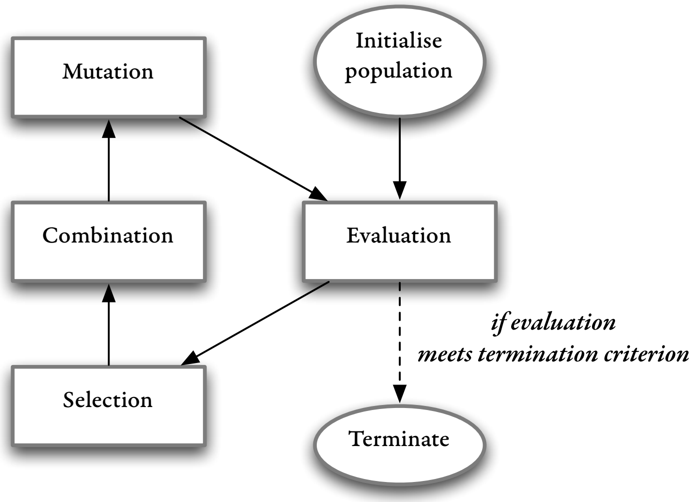

Figure 2.2: Flowchart of a genetic algorithm

spring. Their fitness is determined by a *fitness function* that evaluates how good a candidate solution is for a particular problem.

This process favours evolution towards fitter individuals, mimicking natural selection. The 'breeding' of two individuals usually involves a crossover operation which swaps their information at a randomly selected position. In the interest of maintaining diversification, a *mutation* phase usually occurs after the crossover, to introduce new genetic material into the search and avoid premature convergence on one area of the search space. The mutation operation randomly modifies some information of a selected individual, The process of iterative recombination and mutation, illustrated in Figure 2.2, is repeated until a specific termination criterion is fulfilled.

Application to test data generation

For the purposes of ATDG, the population consists of input test data. The data evolves according to the genetic algorithm in order to satisfy the chosen coverage criteria [23]. In other words, the *fitness function* is based on the *objective function*, the function that evaluates test data according to a certain coverage criterion — the more a candidate contributes to coverage, the fitter it is. Instead of using search heuristics to minimise our objective function, we use it to guide our natural selection process towards the data that will achieve the best coverage.

Regarding the objective function used in the context of genetic algorithms, different variants exist [20]. The *coverage-oriented approach* rewards individuals on the basis of *all* covered program structures. This type of objective function rewards coverage with respect to the overall goal *i.e.* the system is not required to pick a specific path or node and attempt to execute it. The other approach is *structure-oriented* coverage where, similarly to most other search-based ATDG methods, a structure (*e.g.* a path or a node) is selected from the data flow analysis and a separate search is undertaken for each selected structure. Once more, seeing that the *coverage-oriented approach* does not require static analysis of the program under test, it is the most attractive to us for the implementation of Flycatcher.

Finally, the population, individual representation, fitness evaluation and methods of crossover, mutation and selection are all highly dependent on the test data generation problem at hand, hence we will discuss them with regard to Flycatcher when we come to the implementation.

2.2.6 Challenges of dynamic object-oriented languages

Most of the research on ATDG concerns static programming languages [18] and it is only in the past few years that dynamic programming languages have sparked some interest in that field. A possible reason for this is that dynamic programming languages make ATDG harder by enabling features that allow programs to significantly change at runtime. These features can include modifying the type system, extending objects or adding new code during program execution. The challenges this type of behaviour introduces, and that Flycatcher will try and overcome, are listed below [6].

Generating test data of the required type

Given that method parameters do not have static types in dynamically typed languages, we do not know what arguments to pass to them. A potential solution to this is to use a method called *type inference* [27], which tries to infer the type of arguments from the way they are used inside the program. Although this method does not guarantee 100% precision, it is a good starting point for generating accurately typed test data. Mairhofer uses this technique for RuTeG [19], his search-based ATDG tool for Ruby, where the search for test data refines the initially inferred type, by discarding poor candidates.

Generating object instances

Sometimes the type of an object constructor or method parameters will be a complex type and this complicates the test data generation task even further. Generating well-formed object instances to use as arguments inside tests for a dynamically typed object-oriented language is problematic because there isn't a blueprint to construct them from. There is previous work on input data generation for dynamic data structures [15, 37, 30, 38], but all these approaches focus on statically typed languages (C/C++), require static program analysis and mostly lack generality.

Another approach uses something called needed-narrowing [3] or lazy instantiation [16] — where instances are only created when they are actually put to use by the program. This enables test data generators to adjust object instances during execution, when attempts are made to use them, so that they always have the required type. This technique is used by Irulan [1] for generating tests in Haskell, which has lazy evaluation by default. For the purpose of complex type test data generation in Flycatcher, we will investigate whether we can mimic lazy evaluation for JavaScript objects in order to build test data with the appropriate complex type.

Identifying bugs

In dynamic languages such as JavaScript, the function signatures bear no type information and this makes it difficult to know whether an exception is raised due to a wrongly typed test argument or a true program bug. In the case where the exception is not a bug it could be due to two things: manipulating a badly initialised object or breaking a program precondition. The bad initialisation problem might be prevented through the use of lazy evaluation to initialise objects. The breaking of preconditions could be avoided by giving the tester the ability to impose restrictions on the test data generator, so that preconditions for the program are respected. Being able to identify real program bugs during test data generation is an issue that will need further attention during the implementation of Flycatcher.

Dealing with dynamically generated code

Dynamic languages sometimes offer features that parse and evaluate a string at runtime and execute it as code, such as JavaScript's `eval` function. However, not only are these features potentially insecure, they make any analysis for test data generation much harder. As the general use of `eval` in JavaScript is prohibited anyway, we can safely ignore it for the purpose of Flycatcher.

2.3 Object-oriented test case generation

Most of the research on test generation focuses on testing imperative functions, such that the automated generation required is that of the functions' input parameters. However, when dealing with object-oriented code, a different approach is needed, as the unit under test changes from a function to an object. To test one of an object's methods, three steps are necessary [34] and should be repeated until the chosen coverage criterion for the method under test is met.

1. Instantiate the object
2. Call some of its methods to possibly modify its state
3. Assert that the method under test returns the expected answer

Because it is impossible to know how the application will use the class/object under test in reality, as many relevant test cases as possible must be tried in order to maximise the likelihood of finding a bug inside the class in question. Test coverage, the assessment measure that is used for our test case generation problem, is a good indicator of the relevance of test cases for a particular class and method under test. Hence, this measure can be used, as for the generation of input *data*, as a search objective to guide the generation of *method call sequences* for our test cases. Tonella was one of the first to use search-based methods for the generation of adequate OO test cases, using genetic algorithms as the search heuristics method [34]. Indeed, the procedure using genetic algorithms to generate input test data for Flycatcher described in 2.2.5 can simply be extended to the generation of object-oriented tests by adapting individuals so that they represent their structure.

The code example below illustrates the type of object-oriented structural unit test that we aim to generate with Flycatcher (assuming a standard linked list implementation):

```
l = new LinkedList
l.add(true)
l.remove(true)
l.add(true)
assert (l.size == 1)
```

where `LinkedList` is the class under test (a JavaScript prototype) and `size` is the method under test.

2.4 JavaScript

2.4.1 Idiosyncratic features

2.4.2 Object-oriented programming

Chapter 3

Project plan

3.1 Outline

The plan for Flycatcher can be divided into roughly eight stages which are outlined below.

1. **Program Analyser** The Program Analyser component is responsible for retrieving information about the program that is available statically and can be used to generate the test cases. It receives a class or object under test and retrieves the object's methods as well as their signatures. Given that there are no static types in JavaScript, an additional step is needed to try to infer input variables' types from their use in the respective methods. The information collected by the program analyser will be used to create individuals to represent test cases for a particular method under test later on, during the implementation of the genetic algorithm. *This stage of the project is already partially completed.*

2. **Random *Data* Generators for primitive types** Given the added difficulty of generating and initialising object instances to pass in as arguments to the various methods present in our test cases, these will be dealt with at a later stage. At this stage, random data generators will be developed for three of JavaScript's primitive types namely *Boolean*, *Number* and *String*.

3. **Test Case Executor** In order to determine which generated test cases are worth keeping because they are appropriate for the a particular method under test *or* if they contribute to test coverage, an execution environment must be developed. The goal of this environment is to track the coverage achieved by a series of generated test cases, dismissing test cases that are not yet adequate for the method under test or do not contribute to overall coverage. Developing this environment

will most likely involve instrumenting programs under test with code to track the coverage achieved. Depending on their feasibility, either statement or branch coverage[1] will constitute the coverage criterion for Flycatcher and will be tracked by the Test Case Executor.

4. **Random *Test Case* Generator** This step involves building a component that generates unit test cases randomly. The test cases will consist of individuals: a constructor call, a method call sequence ending with the method under test and the appropriate input data at each stage. The test cases that result from this module are bound to achieve poor coverage due to the low probability of exploring deep paths. Hence, this component will be used to benchmark the Flycatcher application.

5. **Search-based *Test Case* Generator** This is probably the core module of the implementation: it consists in generating test cases that strive to achieve good coverage by using heuristic search methods to guide the test case generation. The search heuristics will be effective on two levels: they will direct the generation of *data* for input parameters but also the generation of *method sequences* to be called on the object under test. On top of its evaluation purposes, the coverage feedback from the test case executor will be used by the Search-based Test Case Generator to guide the search for quality input data.

6. **Dealing with complex types** At this stage we will introduce JavaScript complex types (*Array* and *Object*) as potential input parameters in our test generation tool. In an attempt to mimic the concept of lazy instantiation, complex types will be created empty and built up as required by the program. In the case of the Search-based Test Case Generator, the search heuristics will contribute to the creation of object instances of the correct dynamic types (as it did for the creation of correct primitive types).

7. **Extensions** Currently, two possible extensions are envisaged for this project. The first one concerns the introduction of custom input data generators, such that testers can adapt the test data generation to their specific needs in order to achieve better code coverage. Secondly, given that we will use a dynamic approach which comes with a potentially high execution time, another extension would be to investigate and implement methods of speeding up the implementation of Flycatcher. Other extensions are of course possible and may come to light later on during the project.

[1] it is worth noting that branch coverage is the stronger alternative of the two, although its implementation may be more involved

8. **Evaluation** The final stage will consist in evaluating the success of our project and its contribution to the state of the art. The next subsection will describe the intended evaluation process in more detail.

3.2 Evaluation

An evaluation of our system will need to be conducted towards the end of the project, involving experimentation with our automatic test generation tool. The random test case generation will be used as a benchmark for Flycatcher, which uses a search-based solution in order to achieve better coverage. To start off with, experiments will be run on a set of overall relatively straightforward programs of varying size, complexity and number/type of parameters. If the implementation of Flycatcher permits it, we will attempt to use our tool to test larger, real-life software such as JavaScript libraries.

In order to constitute a success, Flycatcher would have to generate unit tests suite for, possibly restricted, JavaScript programs written in an object-oriented style *i.e.* treating JavaScript *prototypes* like classes in the traditional OO paradigm. Additionally, it should perform better using search-based test case generation than random generation and achieve reasonable coverage (statement or branch) compared to other similar applications. We will compare the performance of our tool to applications such as RuTeG which bear resemblances to it, in order to assess its performance from a more general perspective. Overall, the consistent test coverage achieved for programs of various complexities will be the main criterion used in order to evaluate Flycatcher and its contribution to the state of the art in automatic test case generation.

3.3 Time-frame

Our aim is to complete at least steps 1 to 4 in section 3.1 by the end of the spring term. During the spring holidays, not much progress will be made due to a focus on examinations. Hence, of the five weeks remaining after the examination period, the first two will be aimed at completing the section that deals with complex input data types (if it is not finished already), evaluation and looking at possible extensions if time permits. The three final weeks before project submission will be entirely devoted to putting the final report together.

Bibliography

[1] Allwood, T., Cadar, C., and Eisenbach, S. High coverage testing of haskell programs. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (2011), ACM, pp. 375–385.

[2] Anand, S., Păsăreanu, C., and Visser, W. Jpf–se: A symbolic execution extension to java pathfinder. *Tools and Algorithms for the Construction and Analysis of Systems* (2007), 134–138.

[3] Antoy, S., Echahed, R., and Hanus, M. A needed narrowing strategy. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1994), ACM, pp. 268–279.

[4] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (2008), USENIX Association, pp. 209–224.

[5] 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.

[6] Ducasse, S., Oriol, M., Bergel, A., et al. Challenges to support automated random testing for dynamically typed languages.

[7] 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.

[8] Gallagher, M., and Lakshmi Narasimhan, V. Adtest: A test data generation suite for ada software systems. *Software Engineering, IEEE Transactions on 23*, 8 (1997), 473–484.

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

[10] 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.

[11] Gupta, N., Mathur, A., and Soffa, M. Automated test data generation using an iterative relaxation method. In *ACM SIGSOFT Software Engineering Notes* (1998), vol. 23, ACM, pp. 231–244.

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

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

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

[15] Korel, B. Automated software test data generation. *Software Engineering, IEEE Transactions on 16*, 8 (1990), 870–879.

[16] Lindblad, F. Property directed generation of first-order test data. *TFP 7* (2007), 105–123.

[17] LLC, D. Programming language popularity, 2011.

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

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

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

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

[22] Michael, C., and McGraw, G. Automated software test data generation for complex programs. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on* (1998), IEEE, pp. 136–146.

[23] Michael, C., McGraw, G., and Schatz, M. Generating software test data by evolution. *Software Engineering, IEEE Transactions on 27*, 12 (2001), 1085–1110.

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

[25] 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.

[26] 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.

[27] Pluquet, F., Marot, A., and Wuyts, R. Fast type reconstruction for dynamically typed programming languages. In *ACM SIGPLAN Notices* (2009), vol. 44, ACM, pp. 69–78.

[28] Prasanna, M., Sivanandam, S., Venkatesan, R., and Sundarrajan, R. A survey on automatic test case generation. *Academic Open Internet Journal 15* (2005), 1–5.

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

[30] Sai-ngern, S., Lursinsap, C., and Sophatsathit, P. An address mapping approach for test data generation of dynamic linked structures. *Information and Software Technology 47*, 3 (2005), 199–214.

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

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

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

[34] Tonella, P. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes 29*, 4 (2004), 119–128.

[35] Tracey, N., Clark, J., and Mander, K. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)* (1998), pp. 169–180.

[36] Tracey, N., Clark, J., Mander, K., and McDermid, J. An automated framework for structural test-data generation. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on* (1998), IEEE, pp. 285–288.

[37] Visvanathan, S., and Gupta, N. Generating test data for functions with pointer inputs. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on* (2002), IEEE, pp. 149–160.

[38] Zhao, R., and Li, Q. Automatic test generation for dynamic data structures. In *Software Engineering Research, Management & Applications, 2007. SERA 2007. 5th ACIS International Conference on* (2007), IEEE, pp. 545–549.