Flycatcher

Automatic unit test generation for JavaScript

Final Report

Jerome de Lafargue

supervised by
Susan Eisenbach & Tristan Allwood

Department of Computing
Imperial College London

Abstract

The old adage *"to err is human"* is more than manifest in the software world: programmers make mistakes. As a result, it is estimated that 50% of software development time is spent on software testing alone. This has prompted many attempts in the past decades to fully *automate* testing, through tools that generate tests autonomously. In addition to saving time, automatic test generation enables more systematic and unbiased testing, increasing the robustness and quality of programs.

Dynamic languages have recently risen in popularity, as their flexibility suits today's fast-moving software industry. The growth of web applications has led to an increase in the use of JavaScript in particular. Formerly cast as a browser scripting language, JavaScript is becoming a more versatile programming language used for application development as well. This motivates research in automatic test generation for JavaScript, yet the existing work in that domain [19, 2, 38, 5] lacks autonomy and support for the class-based programming style which is becoming the norm in JavaScript development.

In this report, we present Flycatcher, an automatic unit test generation tool written for and in JavaScript. We contribute to the field of automatic test generation by proposing a tool that is capable of successfully generating tests for a comprehensive subset of the JavaScript language. On top of providing the tester with a suite of unit tests, Flycatcher reports the errors found during the test generation process. Experimental evaluation shows that Flycatcher is capable of consistently achieving high code coverage with a selection of benchmark programs.

Acknowledgements

I wish to thank Susan Eisenbach and Tristan Allwood for taking the time out of their busy schedules to discuss this project in our numerous meetings. Susan's extensive DoC experience was highly valuable and Tristan's technical feedback was always very helpful, at times enlightening.

Contents

1 **Introduction** 1

2 **Background** 4
- 2.1 Dynamic software testing . . . 4
  - 2.1.1 Overview . . . 4
  - 2.1.2 Structural testing . . . 5
  - 2.1.3 Unit testing . . . 5
  - 2.1.4 Regression testing . . . 6
- 2.2 Automatic test data generation . . . 6
  - 2.2.1 Overview . . . 6
  - 2.2.2 Static test data generation . . . 7
  - 2.2.3 Hybrid test data generation . . . 8
  - 2.2.4 Dynamic test data generation . . . 9
  - 2.2.5 Challenges of dynamic languages . . . 10
- 2.3 Object-oriented test case generation . . . 11
- 2.4 JavaScript . . . 12
  - 2.4.1 Why JavaScript? . . . 12
  - 2.4.2 Overview . . . 13
  - 2.4.3 Idiosyncratic features . . . 14
  - 2.4.4 Harmony . . . 16
- 2.5 Related work . . . 18
  - 2.5.1 Kudzu . . . 19
  - 2.5.2 Automation of unit testing for JavaScript: prototype . . . 19
  - 2.5.3 JSConTest . . . 19
  - 2.5.4 Artemis . . . 19
  - 2.5.5 RuTeG . . . 20
- 2.6 Summary . . . 20

3 **Overview of** Flycatcher 22
- 3.1 Environment . . . 22
  - 3.1.1 V8 engine . . . 22
  - 3.1.2 Node.js framework . . . 23
- 3.2 Design . . . 23

3.2.1 Components . . . 23
3.2.2 System . . . 24
3.3 JavaScript support . . . 24
3.4 Flycatcher usage . . . 26
3.5 Flycatcher example . . . 27

4 Analysing the source **30**
4.1 Loading . . . 30
4.2 Information retrieval . . . 32
4.2.1 Retrieving class constructors . . . 32
4.2.2 Retrieving class methods . . . 33
4.3 Conclusion . . . 36

5 Generating candidate tests **37**
5.1 Structure . . . 37
5.1.1 Recursive construction . . . 38
5.1.2 Pooling parameters . . . 39
5.2 Types . . . 40
5.2.1 `Unknown` type . . . 40
5.2.2 Type inference . . . 41
5.3 Custom data generators . . . 44
5.4 Outcome . . . 45
5.4.1 Output . . . 45
5.4.2 Timeouts . . . 46

6 Developing a custom runtime environment **47**
6.1 Collecting type information . . . 47
6.1.1 Recording member accesses . . . 48
6.1.2 Accumulating primitive scores . . . 49
6.1.3 Trap threshold . . . 50
6.2 Code coverage . . . 51
6.2.1 Wrapping the MUT statements . . . 52
6.2.2 Implementing the callback . . . 53

7 Evaluation **55**
7.1 Choosing the benchmark suite . . . 55
7.1.1 Triangle types . . . 56
7.1.2 Doubly circular linked list . . . 56
7.1.3 Binary trees . . . 56
7.1.4 Luhn Algorithm . . . 57
7.1.5 Base 64 . . . 57
7.1.6 SHA1 . . . 57
7.1.7 Poker . . . 57
7.2 Experiments . . . 57

7.2.1 Effect of varying the type inference delay 58
7.2.2 Effect of varying the maximum length of tests 58
7.3 Discussion 59
7.3.1 Effect of varying the type inference delay 59
7.3.2 Effect of varying the maximum length of tests 60
7.3.3 Code coverage 60
7.3.4 Limitations 61

8 Conclusion **67**

A Code **69**
A.1 Linked list 70

List of Figures

2.1 Overview of ATDG techniques 7
2.2 The top 10 programming languages 2011 [42] 13
2.3 JSConTest contracts 20

3.1 The Flycatcher system 25
3.2 Flycatcher usage information 26
3.3 Example run 28

7.1 Results for method `Triangle.getType` 58
7.2 Results for method `BinarySearchTree.add` 59
7.3 Results with 1s timeout 64
7.4 Results with 5s timeout 65
7.5 Results with 20s timeout 66

List of Examples

2.1 Object-oriented unit test 12
2.2 JavaScript objects 14
2.3 JavaScript function definitions 14
2.4 JavaScript dynamic types 15
2.5 JavaScript class definition 16
2.6 Object Proxy 17
2.7 Function Proxy 18
3.1 Unit test 29
3.2 Failing test 29
4.1 Runtime dependent class definition 31
4.2 Namespaces 32
4.3 Analyser Proxy's `get` trap 34
4.4 Instantiating a *Function* object 35
5.1 Output format 38
5.2 Executor format 38
5.3 Unreachable code 39
5.4 Pooling 40
5.5 Unknown parameter types 41
5.6 After type inference for `LinkedList.add` 41
5.7 Primitive scoring 43
5.8 Parameter uses 43
5.9 Custom data generators 45
6.1 Fibonacci in JavaScript 50
6.2 Wrapping with burrito 52
6.3 Wrapping in Flycatcher 52
6.4 Wrapped Fibonacci 52

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 [40]. With the fast-paced growth of the software industry, comes the need to test large and 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 [33]. Yet, until recently, dynamic programming languages had mostly been left out of the equation — but their increasing popularity and a renewed interest in them prompts the need to start truly 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 frequently being released for it. 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 [24], 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 thus makes sense to choose it as the object of our work in automatic testing for dynamic languages.

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, and herein lies the main element of risk in this project. On top of that, automatic test generation is not without its challenges. Both the static and dynamic solutions that aim for systematic testing face major hurdles. For the static approach, it is mainly to do with solving the path 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 the challenges listed above 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[2] language but a dynamically-typed one. For instance, our testing tool will need to tackle issues such as the generation of method call sequences and dynamically-typed instances, the latter being intrinsically difficult due to the absence of static types. Our work will thus involve, amongst other things, devising solutions to tackle the well-known challenges that pertain to automatic test generation for *dynamic* languages.

In this report, we present Flycatcher, a program written in JavaScript, that combines existing and innovative methods to achieve automatic generation of unit tests for JavaScript. The word *automatic* is key here, as we believe that to be really useful such a tool requires minimal input from its user. This design choice will therefore guide our decisions throughout, as we strive to create a tool that works autonomously.

Contributions

- The major contribution that this project makes to the field is to extend the limited amount of work done in automatic test generation for dynamic languages by proposing a tool that successfully generates unit test suites for a comprehensive subset of the JavaScript language.

[1] http://nodejs.org/

[2] this has been debated, but JavaScript is heavily object-based and can support polymorphism, inheritance and encapsulation, which we believe is sufficient to call it an object-oriented language

- As well as helping JavaScript developers in their testing effort, we hope that our work will be able to offer new insights into automatic test generation for object-oriented programs in dynamic languages, and benefit future research in that direction.
- Flycatcher also serves to demonstrate the wide-ranging applications of the meta-programming API being put forward by the ECMAScript standards committee. By helping to expose the benefits of the upcoming *Harmony* version of JavaScript, we hope to encourage and speed up JavaScript's move from a quirky scripting language to a fully-fledged programming language.

Report Organisation

The report is organised as follows:

- High-level overview of the Flycatcher application (*Chapter 3*)
- Implementation details (*Chapters 4, 5 & 6*)
- Evaluation and conclusion (*Chapters 7 & 8*)

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, so we will look at the state-of-the-art for doing that too. Finally, we will further justify our choice of JavaScript and describe features of it that are important in 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 [25]. We can characterise it with respect to three parameters: 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 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 [27] 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 used to determine good structural testing is the amount of 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.

Code coverage

Edvardsson lists the most cited criteria [11], from weakest to strongest:

- **Statement Coverage** Each statement must be executed at least once.
- **Branch/Decision Coverage** Each branch condition must evaluate to true and false.
- **Condition/Predicate Coverage** Each clause within each branch condition must evaluate (independently) to true and false.
- **Multiple-condition Coverage** Each possible combination of truth values for the clauses of each conditional statement must be evaluated.
- **Path Coverage** Every single path in the control flow graph 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 statement and branch coverage have been recognised as a basic measure for testing [11].

2.1.3 Unit testing

Unit testing consists in testing individual and independently testable units of source code [31]. 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 they are 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 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 *what we will refer to as* a JavaScript 'class'. Even though JavaScript does not have a class syntax

per se, this is what the units that we will be discussing represent semantically. For clarity, we will therefore from this point on use the word 'class' to refer to them. Exactly what they are, and how they work, will be introduced later in section 2.4.3.

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 [31]. 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

Although object-oriented test generation requires the creation of objects and method call sequences, it shares with procedural test generation, the need for input *data*. Indeed, the object constructors as well as the method invocations require input parameters, hence Automatic Test Data Generation (ATDG) is a key concern to us. As can be seen in Mahmood's systematic review of ATDG techniques [25], many classifications exist for them. For our purposes, the first distinction that we need to make is between white-box, black-box [35] 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 [25, 11, 40]:

1. 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* [11], 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*.

2. The other classification of white-box ATDG concerns the type of implementation: *static*, *dynamic* or a *hybrid* of the two [18, 27]. We will focus on this classification of structural testing as it governs our choice of implementation for Flycatcher. Moreover, the former concerns the target selection stage of ATDG and this step will be ignored in Flycatcher, as it is in many recent ATDG techniques [40]. Figure 2.1 summarises what we believe is an intuitive characterisation of ATDG techniques with respect

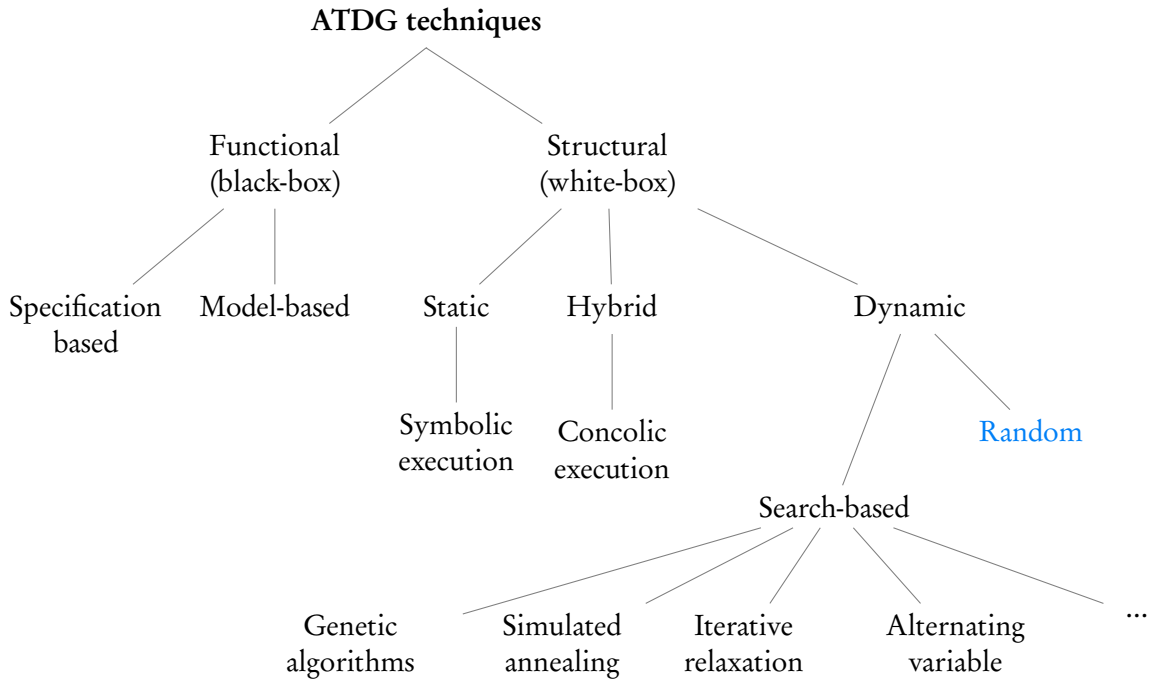

Figure 2.1 – Overview of ATDG techniques

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 only list the most noteworthy to us.

The rest of this section will present in further detail the structural ATDG categories, and the difficulties of ATDG for a dynamic language, so that we can understand the best approach to use when implementing Flycatcher.

2.2.2 Static test data generation

Static test data generation is based on information available from the static analysis of a program, without requiring that the program be actually executed [27]. Static program analysis produces control flow information that can be used to select specific execution paths, in order to try and achieve code coverage of these paths. The goal of static 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, to generate

the test data statically a technique called symbolic execution [21] 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 yields the data necessary for the traversal of that path [20, 21]. There are a lot of technical difficulties associated with symbolic execution [11, 28, 27]:

- the presence of input variable dependent loops can lead to infinite execution trees[1] 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 [36, 16, 32], 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 [33] — we will cover this type of ATDG implementation in section 2.2.3. Due to the numerous problems posed by purely static ATDG, its weakness with dynamic types and constructs [11, 40] and the complexity of building a fully-fledged symbolic executor for a language [11, 18], 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* [33]. 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 — the concrete values are used in these cases. This

[1] the execution paths followed during the symbolic execution of a procedure [21]

type of technique has been popular in recent years, mainly because it overcame the limitations that prevented static ATDG techniques from being applied to industry software. Notable tools that implement it are DART [15], CUTE [39], JPF-SE [3], PEX [41], EXE [7] and KLEE [6].

Yet, although hybrid ATDG deals with some limitations of static ATDG, the constraint-solving based approach that hybrid ATDG also employs is impractical for generating complex input data, which is our aim. Additionally, the highly dynamic nature of JavaScript makes it very difficult to infer any information from the program statically. Hence, hybrid analysis, as it *relies* on static symbolic execution, is not adequate either for our purposes.

2.2.4 Dynamic test data generation

Dynamic test data generation is purely based on actual execution of the software. The program under test is run and feedback is collected at runtime regarding the chosen coverage objective [11]. The feedback is usually obtained through some form of instrumentation of the program that monitors the program flow. Inputs can be generated randomly, relying on probability to achieve the coverage objective — this is known as *random* test data generation [11]. 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 [27]. 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 describe the random and search-based approaches of dynamic test data generation in more detail.

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 has the advantage of being conceptually simple, 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 [11]. In other words, it is difficult for it to exercise 'deep' features of a program that are reachable 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 it is appropriate to use as the *basis* of a new application such as Flycatcher.

Search-based approach

Search-based test data generation uses heuristics to guide the generation of input data, so that the inputs are more likely to 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. Some of the well-known search-based ATDG techniques are *alternating variable* (local search optimisation) [22, 14], *simulated annealing* [45, 44], *iterative relaxation* [17] and *genetic algorithms* [29, 30]. Given that dynamic ATDG is the most suitable strategy for Flycatcher, search-based ATDG is the *optimal* solution. However, the random approach is more straightforward and offers a sensible starting point, so the search-based approach can be seen as a natural extension of that.

2.2.5 Challenges of dynamic languages

Most of the research on ATDG so far concerns static programming languages [25] 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, all during program execution. The challenges that this type of behaviour introduces for ATDG are listed below [9].

Generating test data of the required type

Given that function 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* [34], 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 in a dynamic setting. Mairhofer uses this technique for RuTeG [26], his search-based ATDG tool for Ruby, where the search for test data refines the initially inferred type, by discarding poor candidates. We will inspire ourselves from this approach.

Generating object instances

Sometimes input parameters will be of 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 [22, 46, 37, 47], but all these approaches focus on statically typed languages (C/C++), require static program analysis and mostly lack generality.

Another approach uses needed-narrowing [4] or lazy instantiation [23] — where instances start off as empty placeholder objects and their members are only created when they are actually put to use by the program. This enables test case 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 test data generation in Flycatcher, we will use a new method, but the idea of substitute objects which produce some expected behaviour will be present.

Identifying bugs

In dynamic languages such as JavaScript, the function signatures bear no type information. 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 former can be avoided by ensuring that correct parameters are passed to object constructors *i.e.* the crux of this project. The latter can be solved by giving the tester the ability to impose restrictions on the test data generated, such that the program's preconditions are respected.

As for real software errors, Flycatcher will deal with those too in a way that will be described in later chapters.

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[2], we can safely ignore it for the purpose of our application.

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 only. 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 [43] and should be repeated until the chosen coverage criterion for the method under test is met.

1. Instantiate the object

[2] "https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/eval#Don't_use_eval!"

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 practice, as many relevant test cases as possible must be tried. This maximises the likelihood of finding a bug inside the class in question. Coverage, the assessment measure that is often used for test *data* generation, is an equally good indicator of the relevance of test *cases*. Hence, this measure will be used to assess the quality of test cases as a whole, where coverage is considered for each method of a class independently, as it was for functions.

The code example below illustrates in pseudocode the type of object-oriented structural unit test that we aim to generate with Flycatcher. It assumes a standard linked list implementation, `LinkedList` is the class under test and `size` is the method under test:

```
var l = new LinkedList();
var node = new Node();
l.add(node);
l.remove(node);
l.add(node);
assert(l.size() === 1);
```

Example 2.1 – Object-oriented unit test

The random test *data* generation strategy discussed can simply be extended to the task of generating object-oriented test cases, by randomly selecting the methods (other than the method under test itself) to call on the object under test.

2.4 JavaScript

2.4.1 Why JavaScript?

With cloud computing and the ubiquitous shift of desktop applications to the web, web development has taken on a whole new meaning. Along with this shift, the languages of the web have become much more significant to the software world. JavaScript particularly so, due to its powerful multi-paradigm nature (and despite its unanimously condemned defects). As a result, while in the past JavaScript was used for no more than to animate static HTML pages, today it powers 3D game engines and other fast real-time applications on the web. On top of its pervasiveness on the client-side of the web, it has now also reached web servers and desktop scripting environments.

The move that JavaScript seems to be making from the world of scripts to the world of applications means that it is used in more complex and modular code. This in turn comes with the need to test those applications using standard

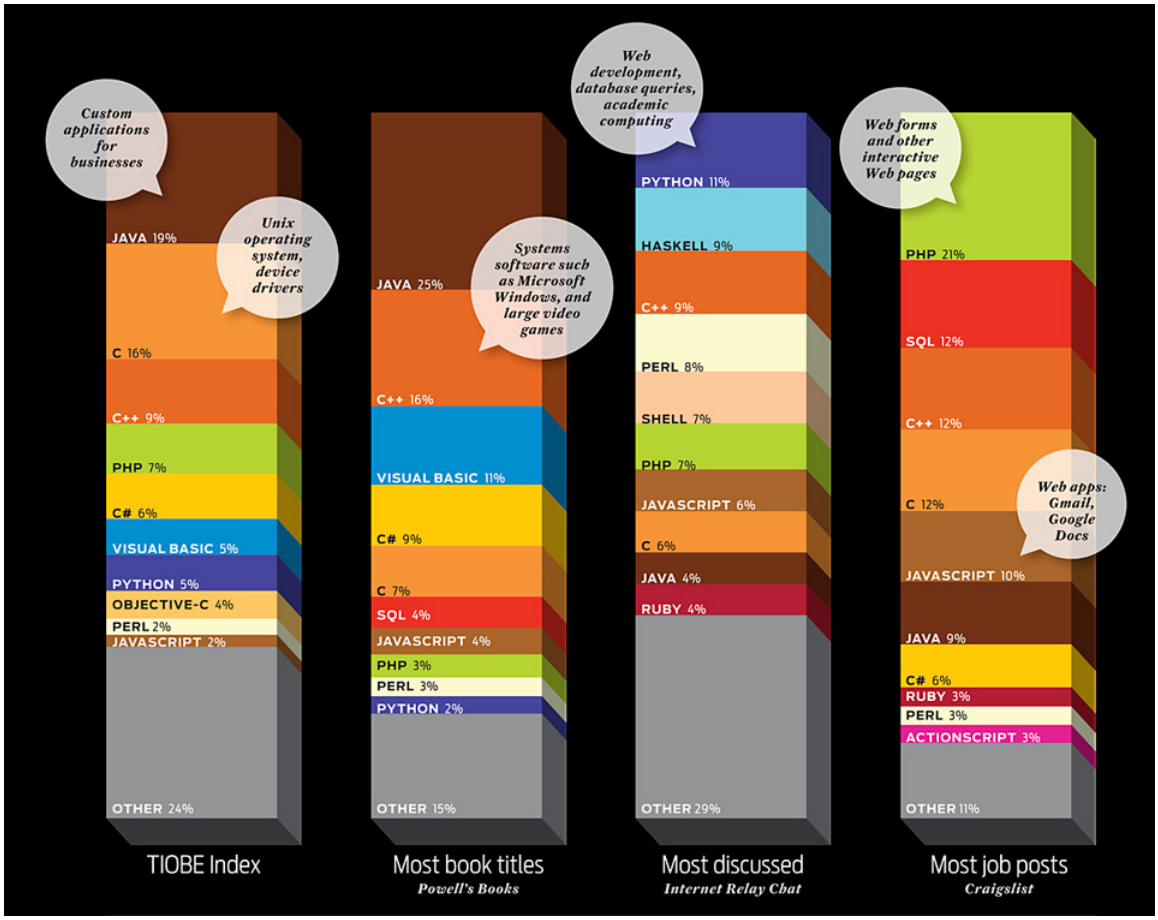

Figure 2.2 – The top 10 programming languages 2011 [42]

software techniques: unit testing, regression testing *etc.* Due to the relative recency of JavaScript's surge in popularity, which can be observed in figure 2.2, not many efforts have been made to automate its testing effort. This is where Flycatcher can make a difference, and this is why JavaScript was chosen for this project: JavaScript is on its way up and there is an opportunity to help its growing community of developers.

2.4.2 Overview

JavaScript is a dynamic scripting language with weak, duck typing and first-class functions. It is multi-paradigm, supporting imperative and functional styles, as well as object-oriented programming, thanks to objects and object prototypes. When used specifically within browsers, JavaScript programs come with certain characteristics, but since we are targeting a wider range of programs, we will focus on the core of the language, as specified by the ECMA-262 5.1 edition

standard [10].

2.4.3 Idiosyncratic features

For the readers who are not familiar with the language, we present the characteristic features of JavaScript [12] that are important in this project.

Types

Types in Javascript can be divided into two categories: *primitive* types and *object* types. The primitive types consist of *Number*, *Boolean* and *String*, as well as *undefined* and *null*, but the latter are particular as they are the only element of their type. Anything else in JavaScript will have the type *Object* (even arrays and functions). Generally, a JavaScript object is an unordered collection of named values, *properties*, that can be stored or retrieved by name *e.g.*:

```
var foobar = { "foo" : 1,
               "bar" : 2 };
foobar.foo     // returns 1
foobar["bar"]  // returns 2
foobar.bar = 3
foobar.bar     // returns 3
```

Example 2.2 – JavaScript objects

JavaScript also has specialised objects, such as *Arrays* and *Functions*. Arrays are untyped, ordered collections of elements (primitives or objects) that can be accessed through a numerical index. Although arrays exhibit additional behaviour, they *can* be thought of as mere objects whose properties happen to be integers.

Functions however, although they are treated as first-class objects and can be stored in variables, differ significantly. They are defined as code blocks with parameters, local scope, an invocation context `this` and may return a value if invoked. Below are examples of a JavaScript function definition:

```
function add(a,b) {
    return a + b;
}
```

or equivalently:

```
var add = function(a,b) {
    return a + b;
}
```

Example 2.3 – JavaScript function definitions

An important feature of JavaScript is that it is dynamically typed, hence a variable cannot imply a type for its value and its type can change over time *i.e.* the code below is correct:

```
var foo = "foo?";
foo = 3;
foo = [1,2,3];
foo = function() {return "foo!"};
foo(); // returns "foo!"
```

Example 2.4 – JavaScript dynamic types

Equally, the function `add` defined earlier does not imply any types for its parameters, it could well be a function that concatenates two strings for instance. Although JavaScript supports the object-oriented paradigm, the way it does so is quite different from most object-oriented languages and we will see that functions play a central role.

Object-oriented programming

JavaScript is known as a *prototype-based* language, meaning that it does not have traditional class definitions to represent object blueprints and create instances of them. Instead, object instances can be used as prototypes to construct other objects instances. That way, we can say, tying this back to classical OOP, that if two objects inherit properties from the same prototype object, they are instances of the same class. The role that functions have in this is that, usually, if two objects inherit properties from the same prototype object, it means that they have been created and initialised by the same function — this function is known as their *constructor*.

A constructor is a function typically *designed* for the initialisation of newly created objects. When called with the keyword `new`, its invocation context represents the object being created, hence it can initialise the object's properties by using the `this` keyword and then return it as the newly created object. A key feature of constructors is that their `prototype` property (an object) is also used to initialise the object they construct — the new object *inherits* from the prototype object which is why it is called prototypical inheritance.

To summarise, a class in JavaScript is defined by a constructor function[3] through two elements:

1. its body, which initialises objects through accessing `this` (the class fields are defined here)
2. its `prototype` property, from which the constructed objects inherit all properties (the class methods are defined here)

[3] which is not technically different from standard functions except that it is *intended* to be called with the `new` keyword

Below is an example of a simple class definition for a circle, to illustrate our explanation:

```
function Circle(radius) {
    // the Circle class has a field named radius
    this.radius = radius;
}

// the Circle class has a method getCircumference
Circle.prototype.getCircumference = function() {
    return this.radius * Math.PI * 2;
}

var c1 = new Circle(1);
var c2 = new Circle(2);

c1.getCircumference() // returns approx. 6.28
c2.getCircumference() // returns approx. 12.56
```

Example 2.5 – JavaScript class definition

In the example above, note that the invocation context of the constructor is returned automatically when the constructor is called with the keyword `new`. Also, both `c1` and `c2` have a `radius` field (though initialised to different values) and a `getCircumference` method, as they were initialised with the same constructor — we can say that they have the same 'class'. In Flycatcher, classes are what we will use as the target unit for generating suites of unit tests and this is what we will be referring to.

Finally, although classical OOP techniques such as subclassing, polymorphism and encapsulation are all possible in JavaScript, they are not a key concern to us and we will therefore not discuss them.

2.4.4 Harmony

The JavaScript language and its derivatives JScript, ActionScript *etc.* were formalised in 1997 by Ecma International under the name ECMAScript. The ECMAScript specification, namely ECMA-262, standardises the core of the JavaScript language and thus serves as a common ground for its implementation. The latest published version of the standard is edition 5, which is implemented in all major browsers.

Nevertheless, there is a very interesting and landmark edition in progress called *Harmony*, with many new exciting language features. Among those features is a meta-programming API, which, although currently non-standard, is already implemented by major browser engines such as Google's V8 engine and Firefox's SpiderMonkey. The meta-programming API presents a new type of object which is extremely powerful and helpful for the development of Flycatcher: the *Proxy* object.

Object Proxies

Proxies are special objects that let the programmer define the *behaviour* of an object *i.e.* how it responds to low-level operations. This is done through a 'catch-all' mechanism, which *traps* or *intercepts* low-level operations on the proxy and allows us to redefine their outcome. To illustrate this, we define a simple *Proxy* that overrides the `[[Get]]` low-level operation and traps property accesses of the form `proxy.name` in the context of *getting* a property's value. We override that behaviour and return the string `"It's a trap!"` instead:

```
var proxy = Proxy.create({
  get: function(receiver, name) {
    return name + " -> It's a trap!";
  }
});

proxy.treasure = "gold";
proxy.treasure // returns "treasure -> It's a trap!"
```

Example 2.6 – Object Proxy

The `receiver` is a handle to the proxy itself, and the `name` is the name of the property being trapped. The object passed to the *Proxy*'s `create` function is called its handler and needs to implement a specific API, so that all of the fundamental low-level operations on the object respond. The traps available to Proxies and the code they respectively emulate are laid out in table 2.1. The *fundamental traps*' implementation is required, but the *derived traps* can be left out as they have a default implementation in terms of the fundamental ones. In the context of Flycatcher, the traps that we will be the most interested in are `get` and `set`.

It is worth noting about *Proxies* that some operations are not trapped in order to respect the language invariants:

- The tripple equal `===` operator isn't trapped *i.e.* `p1 === p2` *only* if `p1` and `p2` are a reference to the same *Proxy* object
- The `typeof` operator
- The `instanceof` operator
- The `Object.getPrototypeOf(proxy)` operation

There is much more to say about object *Proxies* and their applications, but this brief introduction is sufficient in our case. The full proposal can be found online [8].

Trap	Emulated code
Fundamental Traps	
`getOwnPropertyDescriptor(name)`	`Object.getOwnPropertyDescriptor(proxy,name)`
`getPropertyDescriptor(name)`	`Object.getPropertyDescriptor(proxy,name)`
`getOwnPropertyNames()`	`Object.getOwnPropertyNames(proxy)`
`getPropertyNames()`	`Object.getPropertyNames(proxy)`
`defineProperty(name,pd)`	`Object.defineProperty(proxy,name,pd)`
`delete(name)`	`delete proxy.name`
`fix()`	`Object.[freeze|seal|preventExtensions](proxy)`
Derived Traps	
`has(name)`	`name in proxy`
`hasOwn(name)`	`Object.prototype.hasOwnProperty.call(proxy,name)`
`get(receiver,name)`	`proxy.name`
`set(receiver,name,val)`	`proxy.name = val`
`enumerate()`	`for(prop in proxy)...`
`keys()`	`Object.keys(proxy)`

Table 2.1 – Proxy traps

Function Proxies

Harmony also proposes *Function Proxies*, which are useful to us. Functions in JavaScript are objects, hence, *Function Proxies* have the *same* trapping capabilities but they also offer *additional* traps, which are specific to functions: the `call` trap and the `construct` trap. For illustration, we give an example of a *Function Proxy* in action:

```
// handler is as earlier, construct and call are functions
var fnproxy = Proxy.create(handler, call, construct);
fnproxy.treasure   // calls handler.get(fnproxy, treasure)
fnproxy(1,2,3)     // calls call(1,2,3)
new fnproxy(1,2,3) // calls new construct(1,2,3)
```

Example 2.7 – Function Proxy

2.5 Related work

Flycatcher is the first tool to automatically generate unit test suites for JavaScript programs written in an object-oriented style, where classes are the target unit. In this section, we discuss projects that have applied different techniques for the generation of tests in JavaScript, as well as some similar work in a different programming language.

2.5.1 Kudzu

Kudzu [38] is a tool that, given a URL for a web application, automatically generates high-coverage test cases to systematically explore its execution space. The aim of the tool is to uncover security vulnerabilities in the browser such as client-side code injection. It uses symbolic execution, based on a custom constraint solver that supports the specification of boolean, machine integer and string constraints, including regular expressions, over multiple variable-length string inputs.

This automated vulnerability analysis tool significantly differs from Flycatcher in that it specifically targets *client-side* code and in-browser interactions, focusing heavily on string inputs. The test generation method used belongs to the category of *hybrid* ATDG, which also differs from the method of choice for Flycatcher: *dynamic* ATDG.

2.5.2 Automation of unit testing for JavaScript: prototype

A prototype has been developed by Mohammad Alshraideh [2], with a very similar purpose to that of the Flycatcher application: automatically generate unit tests for JavaScript programs. However, the philosophy behind it differs significantly. To use the proposed tool, the tester *has* to annotate class files in order to specify:

- the sequence of method calls in tests
- the type and range of parameter values

Moreover, only primitive parameters are handled. This tool's contributions are thus far from Flycatcher's objectives of handling object-oriented code in an *autonomous* manner.

2.5.3 JSConTest

JSConTest [19] is a test case generator for JavaScript that uses *contracts* to generate input data for the tests. The contracts are type signature annotations that the tester has to include in his program, to enable systematic verification of programs despite the absence of static types. Figure 2.3 gives an example of the contracts in JSConTest.

2.5.4 Artemis

Artemis [5] is a framework for automated test generation in JavaScript which targets scripts in HTML pages. This tool uses feedback-directed random test generation, meaning that execution is monitored to collect information that directs the test generator towards input that yields increased coverage. However, this

```
1  /** int → int */
2  function f(x) { return 2 * x; };
3
4  /** (int,int) → bool */
5  function p(x,y) {
6    if (x != y) {
7      if (f(x) == x + 10) return "true"; // contract violation
8    };
9    return false;
10 };
```

Figure 2.3 – JSConTest contracts

tool targets a different class of programs than Flycatcher as it focuses on web-specific features such as AJAX, the Document Object Model and its event-driven execution.

2.5.5 RuTeG

RUTEG [26] is an automatic unit test generation tool for the dynamic programming language Ruby. Although it targets a different language, this work is worth mentioning here as the following characteristics are also found in Flycatcher:

- it generates suites of structural unit tests
- it targets a dynamically-typed language
- it generates object-oriented tests
- it can handle complex input data

2.6 Summary

Despite being tedious and prone to human error, testing is a necessary and important part of software development — it is thus worthwhile to attempt to automate that effort where possible. In this project we focus on structural testing: making sure that the *internals* of an application work by trying out, as much as is feasible to do so, all the ways in which it can be executed. The quality measure for our tests is therefore code coverage: how much code we can, with fair confidence, assert to be bug-free.

In the arena of automatic test generation, dynamic languages have so far largely been left aside. Given a recent surge in JavaScript's popularity, a growth of its developer community, as well as an expansion towards server-side application development, we feel that it makes sense to spend time improving the research work on automatic test generation for this language. From the wide array of techniques in this domain, we will use the most appropriate for JavaScript,

namely dynamic test generation (as opposed to static or hybrid): test generation that results from numerous executions of the program and their feedback. Within dynamic test generation, the random approach is the method of choice for Flycatcher as it represents a natural starting point for a new implementation. Finally, we saw that there are challenges that pertain specifically to object-oriented languages and dynamic languages and this project will involve devising novel ways to overcome these issues in the context of JavaScript, using state-of-the-art features of this language.

Chapter 3

Overview of Flycatcher

The development of Flycatcher can be divided into distinct phases, which correspond to the components of the application. In order for the reader to follow and understand the development process, we feel that it is best to start off by giving them a sense of the big picture. Hence, in this chapter we will explain our choice of programming environment, briefly describe the development stages, give an overview of the system, as well as an example of Flycatcher in action.

3.1 Environment

3.1.1 V8 engine

In picking a JavaScript engine to work with to develop Flycatcher, we looked for the following characteristics:

- developer friendliness
- a standalone release (many are coupled with browsers)
- speed of execution
- open source
- strong online community
- conform to the latest ECMAScript standard, ECMA-262 edition 5
- meta-programming features
- runs on x86 or x86-64 processors

Of the three main contenders in these categories, namely Firefox's SpiderMonkey and Rhino engines and Google's V8, V8 was chosen as it was by far the strongest, notably in terms of execution speed and developer friendliness. The version of V8 used is 3.9.5.

3.1.2 Node.js framework

JavaScript's debut on the server side prompted the need for a form of application development library support for the language. The development of Node.js that started in 2009 was an attempt to satisfy this need. Although the framework is intended as an event-driven web framework, the fact that it has a strong online developer community and a variety of valuable open source contributions makes it appealing for developing JavaScript in general. It is all the more appealing to us because it is built on top of the V8 JavaScript engine, which is our engine of choice.

On top of its built-in library support, Node.js offers an efficient package manager `npm`, which allows us to effectively separate our work into components, as well as easily import plugins from the open-source community. The Node.js release used to deveop Flycatcher is version 0.7.5.

3.2 Design

The process of automatically generating tests using the approach we have chosen, dynamic test generation, naturally divides into distinct stages. In this section we will outline and briefly describe what those stages are and introduce the components of the application that they correspond to.

3.2.1 Components

Analysing the source

The very first task that Flycatcher needs to perform is a dynamic *analysis* of the source code, in order to extract information about the program under test. The Analyser component performs this role: extracting the information that is necessary to even start the test generation process at all. Intuitively, we can think of it as mapping the source code into Flycatcher's data structures, which are then used in the test generation process.

Generating candidate tests

The following stage consists in generating *candidate* or *eligible* tests. These generated tests are run inside a custom runtime environment and depending on the *feedback* provided by that environment, they may be eligible to become part of the final suite of unit tests which is output to the user. The early tests are not accurately typed and therefore not eligible, but serve to gather runtime information until we have enough information to generate tests that are. As such, the

Test Generator component which fulfils this role is tightly coupled with another: the one which orchestrates a virtual runtime environment in order to collect the necessary runtime information.

Developing a custom runtime environment

The custom runtime environment which is necessary to run candidate tests and provide feedback on their eligibility, is implemented by the Executor component. Its main responsibilities are:

1. enabling the collection of information concerning the type of method parameters
2. tracking the code coverage achieved by candidate tests, to assess their quality

The Executor and the Test Generator thus work in to-and-fro until either full code coverage is achieved or a termination criterion is met. Upon termination, the tests that are deemed accurately typed and that contributed to code coverage are collated into a suite of unit tests. They are then output to the user in a format corresponding to his preferred unit testing framework. The tests that reveal an error in the program under test or a possible mistake made by Flycatcher's type inference mechanism are also output, as *failing tests*.

3.2.2 System

Figure 3.1 gives an idea of the overall system and how the components fit together, so that the reader can appreciate the journey from the program under test to a suite of test cases that can be used for regression unit testing.

3.3 JavaScript support

In this section, we elaborate on which JavaScript features are supported by Flycatcher. First of all, there are two separate matters to consider:

1. the support for JavaScript inside the classes being tested
2. the support for JavaScript concerning the types of parameters

Given that Flycatcher is built on one of the latest versions of the V8 engine, it supports all of the ECMAScript Edition 5 constructs *inside* the classes under test.

However, there are limitations when it comes to the support for the *parameters*' types, due to the fact that these types will need to be inferred by Flycatcher, and there are restrictions as to what types *can* be. Flycatcher supports the inference of the primitive types *number*, *string* and *boolean* and therefore also their

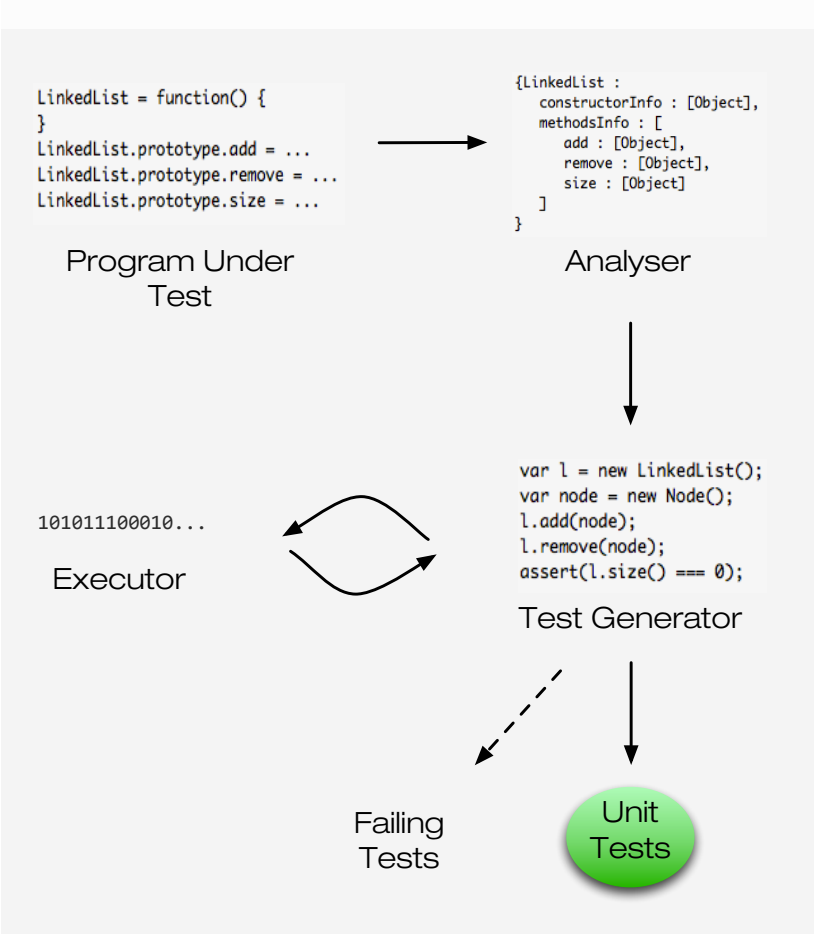

Figure 3.1 – The Flycatcher system

object counterparts *Number*, *String* and *Boolean*. Additionally, Flycatcher supports used-defined types, as long as their definition is accessible in the source code under test. However, it is worth noting that there is currently no support for other standard objects, including *Array* and *Function*, due to the technical challenge they represent. Finally, there is no support for the parameters to be literal objects *i.e.* objects must belong to a class, which is in line with the object-oriented style targeted by this tool. This is also appropriate in the context of unit tests, as they are meant to test the external interface of a self-contained unit. The elements of that interface must therefore be of a well-known primitive or user-defined type. Nevertheless, the lack of support for *Array* and *Function* does have its drawbacks, which we will discuss in the evaluation chapter.

3.4 Flycatcher usage

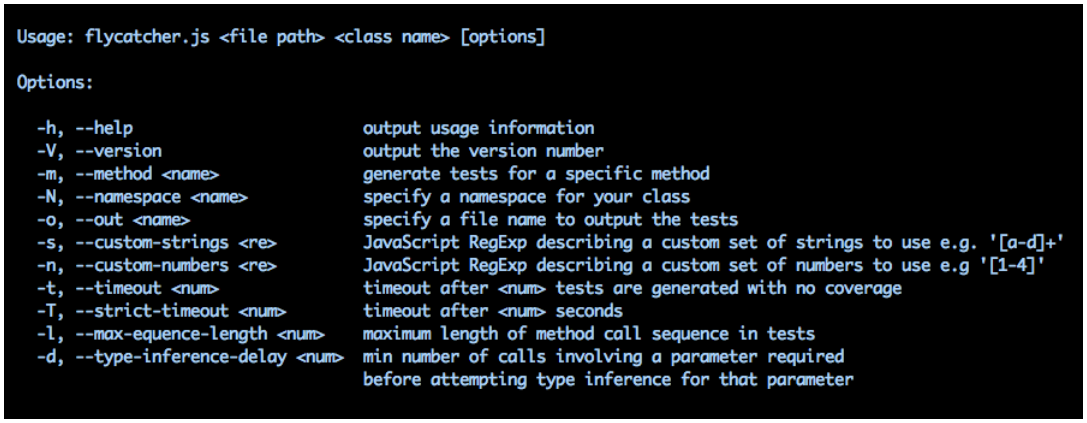

Figure 3.2 – Flycatcher usage information

Flycatcher's usage information can be seen in the screenshot in figure 3.2. The required arguments are the source file where the class under test can be found as well as the name of that class. The following arguments are optional:

- `--method <name>`: Specifying a single method from the class under test, tests are only generated for that method. If this is not specified, tests are generated for all of the class under test's methods.
- `--namespace <name>`: This option serves to specify a namespace inside which to look for the class under test in case it is not available in the global context.
- `--out <name>`: Specifying a file for the tests to be output. The unit test suite is output in `<name>.js` and the failing tests are output in `<name>.log`. If this is not specified the tests are output in a default file.
- `--custom-strings and --custom-numbers <re>`: So that the random input strings and numbers that are generated by Flycatcher for the unit tests are *appropriate* for the program under test, custom generators can be specified for each. These are specified by a string `<re>` which represents a regular expression that conforms to the JavaScript RegExp syntax. For example, to generate only strings with characters between *a* and *f*, one would use `"[a-f]+"`.
- `--timeout <num>`: Indicates termination after `<num>` *valid*[1] tests are generated without achieving any coverage. This number indicates that the part of the code that remains to be covered is either unreachable or deeply

[1] tests for which we have already inferred types for all parameters involved

nested. However, in the event that it is the former, a certain termination criterion is decided upon.

- `--strict-timeout <num>`: The `--timeout <num>` adapts to the program under test *i.e.* if it is a program that performs long calculations this is taken into account. However, if a precise deadline is needed, it can be specified in seconds with the `strict-timeout` option.

- `--sequence-length <num>`: Method sequences in tests are generated by Flycatcher at random, but in order for tests to have a reasonable size, the sequences must have a maximum length. This maximum can be specified with this option, which defaults to ten if none is specified (based on experimentation).

- `--type-inference-delay <num>`: Before types are inferred for parameters inside candidate tests, a number of tests are run inside Flycatcher for the sole purpose of collecting information on those parameters. Every time such a temporary test is run, we keep track of the parameters that are used for all of the functions involved. This gives us a measure of how much information has potentially been collected for each parameter *i.e.* how confident a type estimate we will be able to make for that parameter at any point in time. This advanced option sets the number of parameter uses after which we consider having enough information for type inference. Note that it requires the user to understand the way Flycatcher works internally. So that most users don't have to, the variable defaults to a high value (the value is based on experimentation), in order to avoid making unsuccessful type inferences.

3.5 Flycatcher **example**

In figure 3.3, we give an example of Flycatcher running with the default settings for the method `remove` of the `LinkedList` class (see Appendix A.1), a standard linked list implementation. Parameters are inferred for all of the class's methods, not just the one under test, as those methods are used inside the test cases. As the command line output shows, in the early stage of the test generation process, the correct types are inferred for all the parameters when possible.

In a linked list implementation, it is often the case that the nodes' values are not used inside the implementation itself — they are only used in the context where the linked list is used. Hence we cannot collect any information regarding the type of the `data` parameter and in order for the tool to retain as much autonomy as possible, it is substituted with a random type, here a *string*. From the example we can see that 100% test coverage is achieved for the `remove` method of the `LinkedList` class, thanks to the generated unit tests like the one in example 3.1. Tests that fail even though all parameter types have been resolved indicate a

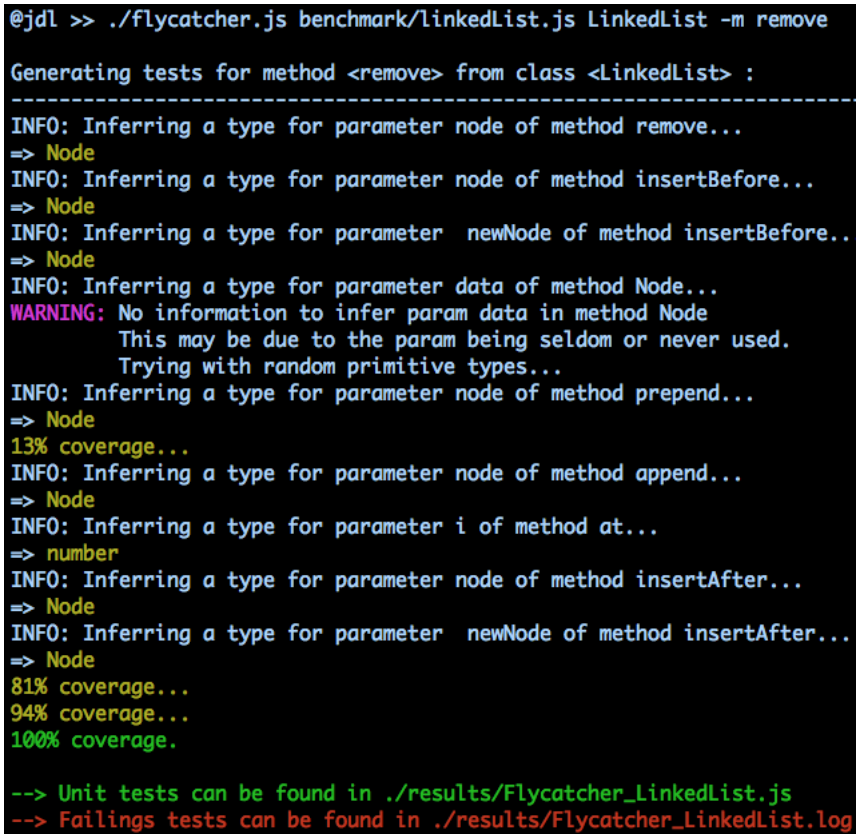

Figure 3.3 – Example run

weakness in the program. For instance, in example 3.2 we can see that the linked list implementation does not check when calling `insertBefore`, whether the target node exists in the list, which gives rise to a `TypeError`.

```
var linkedlist325 = new LinkedList();
var node326 = new Node("AdlwkJCa2");
linkedlist325.prepend(node326);
assert.equal(linkedlist325.remove(node326), true);
linkedlist325.size();
var node327 = new Node("5asd24all");
linkedlist325.prepend(node327);
var node328 = new Node("azcma5");
assert.equal(linkedlist325.remove(node328), false);
// Success!
```

Example 3.1 – Unit test

```
var linkedlist309 = new LinkedList();
var node310 = new Node("J619y4xu1");
var node311 = new Node("segBPUR5");
linkedlist309.insertBefore(node310,node311);
linkedlist309.at(22);
var node312 = new Node("L93Ifj7t75");
linkedlist309.append(node312);
var node313 = new Node("9");
linkedlist309.remove(node313);
// TypeError: Cannot set property 'next' of null
```

Example 3.2 – Failing test

In the next three chapters, we elaborate upon this high-level overview, by detailing the implementation of the three core stages described:

1. Analysing the source
2. Generating candidate tests
3. Developing a custom runtime environment

Chapter 4

Analysing the source

The targets of our unit tests are classes, understood as defined in section 2.4. We refer to the class for which tests are being generated as the *class under test*, using the acronym CUT. Because of the nature of meaningful tests in an object-oriented context, generating tests for a class means generating tests that target individual methods of that class, as shown in section 2.3. These methods, when selected one at a time for the purpose of generating test cases, will be referred to as *method under test* or MUT.

The Analyser component is responsible, intuitively, for retrieving information regarding the CUT and its methods, and storing that information in convenient data structures, so that they can be accessed by the rest of the Flycatcher application. This is because that initial information not only serves as the starting point for the test generation process, but is used and updated throughout that process.

4.1 Loading

First of all, it is important to note that, given the highly dynamic nature of JavaScript, it is *necessary* for us to execute or load the class definition. In other words, a static analysis of the source code will not be able to give us the information that we need about the class under test, namely:

- its constructor definition
- its method definitions
- its class fields

To illustrate why this is impossible, consider this class definition:

```
var LinkedListConstructor = function() {
    this.size = function() {
        // implementation
    }
}

var LinkedList = LinkedListConstructor;

var realAddFunction = function(node) {
    // implementation
}

var addFunction = realAddFunction;

var getRemoveFunction = function() {
    return function(node) {
        // implementation
    }
}

LinkedList.prototype.add = addFunction;
LinkedList.prototype.remove = getRemoveFunction();
```

Example 4.1 – Runtime dependent class definition

The example is unnecessarily but purposely entangled, to demonstrate the point. First, the class constructor is not defined directly but is in fact a reference to another function. This is possible in JavaScript because functions are first-class objects. Second, the `LinkedList` class has three methods `size`, `add` and `remove` but each are defined in a different manner, and sometimes also through indirect references.

Unlike a language like C++ or Java that have static class definitions which follow a predictable format, in JavaScript class definitions cannot be learnt statically. The example above could be made much more complex and still be a correct class definition, and to try and learn class definitions statically while covering all possible scenarios would effectively amount to executing the class definition. In other words, simply parsing the source code does not get us very far. Hence, this means that we need to extract the information we need *dynamically*, and this involves two steps:

1. Loading/executing the source code to obtain the *constructor* of the CUT
2. Using that constructor to *instantiate* the class in order to obtain the MUTs

Thankfully, Node.js's built-in library offers a virtual machine API which enables us to do just that. In JavaScript, any variable defined at the outermost scope in a source file becomes a property of the *global object* or *context* when that file

is interpreted. The Node.js virtual machine API exposes:

```
vm.runInNewContext(code, [sandbox])
```

This means that we can load the source file under test using standard IO, and execute it as `code` with a fresh global object `sandbox`. This means that we do not pollute the environment in which we interpret the class definition with Flycatcher's own global object. It also means that through `sandbox` we have an unambiguous handle on the objects that form the class definition of the CUT.

In fact, when the vm has finished interpreting the source under test, we have access not just to the CUT, but to all the other classes which are accessible in that scope. These classes are significant since they are *potential candidates* for the user-defined types of the parameters of the CUT constructor and methods. For instance in the running `LinkedList` example, it wouldn't be surprising to find a `Node` class in the `sandbox` as well.

Note that all of these classes, including the CUT, may belong to a namespace like so:

```
DataStructures.LinkedList = function () {
}

DataStructures.LinkedList.prototype.add = function(node) {
   // implementation
}

etc.
```

Example 4.2 – Namespaces

To overcome this, Flycatcher gives the option of specifying the namespace of interest, but the underlying mechanism for extracting information remains the same.

It is necessary for any classes that are used by the CUT to be accessible in the provided code and namespace, or else Flycatcher will not be able to generate tests with the correct parameter types.

4.2 Information retrieval

4.2.1 Retrieving class constructors

Once the source code under test is loaded into the sandbox, Flycatcher iterates through the appropriate namespace and retrieves the properties which have the type *function*. These are *all* potential class constructors as JavaScript constructors are no different than ordinary functions. The name of the CUT will have

been specified by the user, which enables us to give that function a special status in our data structures when we retrieve properties from the sandbox.

Information about all of the classes of interest will need to be used and updated throughout the course of the test generation process. Hence, while iterating through the sandbox, we initialise a convenient data structure in order to store that information. This structure, the `ProgramInfo` object, acts as a placeholder for all of the information about the program under test that is relevant to the test generation process.

4.2.2 Retrieving class methods

Given the sandbox, retrieving a class's constructor is straightforward but to retrieve the class's methods dynamically, as is required, instances of it need to be created[1]. This requires us to be able to:

1. create parameters that will not crash the constructor
2. use them to initialise the class

Creating parameters that will not crash the constructor

At this point we may want to remind the reader that *JavaScript function parameters bear no type information whatsoever*. Hence, at this stage, we have no idea what parameters to create to pass to the constructor. If we do pass wrong parameters to it however, such as a number when an object is expected, the constructor will crash and we will not be any closer to retrieving information about a class's methods.

This leads to our first use of the Proxy object proposed by ECMA-262 Harmony, introduced in section 2.4.3. In order for the constructor not to crash, we pass Proxy objects to it, which have the ability to respond to any operation: Proxies that return other Proxies. Hence, even if property accesses are made on the *result* of a property access, the constructor will continue executing, as the `get` trap of the Proxy returns a reference to itself. However, we must account for the fact that the trapped function may expect *a function* in return, and we must therefore return a *Function Proxy*. The Function Proxy has the same behaviour as the Proxy but can *additionally* trap attempts to invoke or instantiate it.

The catch is that primitive operations are also trapped by the `get` trap, as they translate into the function `valueOf` being called. If we return a Proxy when `valueOf` is trapped, the engine will try to apply a primitive operator, such as `++`, to an object, and throw a type error. Hence, the Proxy's `get` trap must return a *primitive* whenever the method trapped is `valueOf`. Returning a number is a sound choice as it does not make any of the primitive operators effect a crash (but for instance `"string"++` would). The Proxy API offers many more traps than

[1] `MyClass.prototype`, accessible from the context, will give access to some of a class's methods, but others may also be defined inside the class constructor

the `get` trap, but we focus on the `get` trap for clarity's sake, as the other traps simply implement idle behaviour. To summarise, we lay out the implementation of the `get` trap that we have explained:

```
get: function(rcvr, name) {
    var proxyHandle = this;
    if (name === "valueOf") {
        return function() {
            return 1;
        }
    }
    else {
        return Proxy.createFunction(proxyHandle, // Object Proxy
                                    function() { // Function Proxy
                                        return Proxy.create(
                                            proxyHandle)
                                    });
    }
}
```

Example 4.3 – Analyser Proxy's `get` trap

It is worth noting that we do not care about the *outcome* of the operations on parameters within the constructor, only that it does not crash. The purpose of this process is to find out the signatures of a class's methods and these cannot be affected by the constructor's parameters — except for the case where *the class method itself* is passed in as a parameter, which we do not deal with (we do not handle *Function* type parameters in general).

Tests were run to ensure that the constructor does not crash with the Proxy parameters created. The test cases are summarised in table 4.1, where `proxy` is the parameter proxy, `f` is a field and `m` is a method:

Using the proxy parameters to initialise the class

Once the appropriate number of Function Proxies have been created, which we can find out from the retrieved constructor's definition, we need to initialise the class with them. One would think that JavaScript, given its first-class functions, would have a way to do that — we have the constructor function and we have the parameters for it. Unfortunately, the `Function.apply` and `Function.call` library functions both simply *invoke* functions, they cannot *instantiate* a new object with them, which is what we want to do. Thankfully, this is easily resolved with a small closure:

Operation	Code
`valueOf` call on proxy itself	`proxy + 1`
`toString()` call on proxy itself	`proxy.toString()`
Field access	`proxy.f`
Field access of field access	`proxy.f.f`
Method call of field access	`proxy.f.m()`
Overriding field	`proxy.f = 3`
`valueOf` call on field	`proxy.f + 1`
`toString` call on field	`proxy.f.toString()`
Method access	`proxy.m`
Method call	`proxy.m()`
Field access of method call	`proxy.m().f`
Method call of method call	`proxy.m().m()`
Overriding method	`proxy.m = function(){}`
`valueOf` call on method	`proxy.m.valueOf()`
`toString` call on field	`proxy.m.toString()`

Table 4.1 – Testing proxy parameters

```
// ctr is the constructor of the class we want to instantiate
var construct = (function() {
    function Copy(args) {
        return ctr.apply(this, args);
    }
    Copy.prototype = ctr.prototype;
    return function(args) {
        return new Copy(args);
    }
})();

// proxyParams are the Function Proxy objects described
var instance = construct(proxyParams);
```

Example 4.4 – Instantiating a *Function* object

Once we have access to an instance of a class we can just iterate through its properties to obtain its methods (the functions) and its fields (the rest). Moreover, this technique fully supports JavaScript polymorphism, as any superclass methods will be inherited when the subclass is constructed and thus present amongst the properties of the available instance.

4.3 Conclusion

At this point, the Analyser has successfully built an object that embodies the structure and information of all the classes in the scope of the program under test. For clarity's sake, we will refer to that object throughout by naming it `ProgramInfo`. To summarise, this is the information that has been gathered so far for each class:

- constructor definition ✓
- method definitions ✓
- class fields ✓

However, as the `ProgramInfo` object is the central element in Flycatcher, it has additional responsibilities. But these will be revealed in the later implementation chapters, where relevant.

Chapter 5

Generating candidate tests

In this chapter, we detail the process by which Flycatcher generates random candidate tests, in the hope of eventually generating some that can be used to achieve good code coverage of the CUT. First, we explain the structure of a test and some characteristics of that structure. Then we describe the mechanism for trying to generate tests in which the types of the parameters are accurate. Finally, we introduce the concept of custom data generators, for seeding primitive type values inside our tests.

5.1 Structure

To recapitulate, an object-oriented test targets a particular method of the CUT, referred to as the MUT. If the method of interest is specified by the user then tests are only generated for that method. Otherwise each of the CUT's methods are selected as the MUT, and tests are generated for each of them, covering the whole class. Either way, generating a test for a MUT involves the following steps:

1. Create an instance of the CUT
2. Call some of its methods (including the MUT) to possibly modify its state
3. Where the MUT is called (at least once), assert that it returns the expected answer

Assuming that the MUT is the method `size`, the final output of a valid unit test might therefore look like:

```
var linkedList = new LinkedList();

var node1 = new Node(123);
linkedList.add(node1);

var node2 = new Node(234);
linkedList.add(node2);
assert(linkedList.size() = 2);

var node3 = new Node(345);
linkedList.add(node3);
assert(linkedList.size() = 3);
```

Example 5.1 – Output format

Note that the format shown in example 5.1 differs from the format of tests that are destined to be run in the Executor. The same test in the Executor needs a structure to keep track of MUT call results, in order to later construct assertions if the test is selected. Example 5.2 illustrates the format of tests in the Executor. In the rest of the examples however, we omit the anonymous function and the `results` structure to keep the examples clear and concise.

```
(function() {
   var results = [];
   var linkedList = new LinkedList();

   var node1 = new Node(123);
   linkedList.add(node1);

   var node2 = new Node(234);
   linkedList.add(node2);
   results[0] = linkedList.size();

   var node3 = new Node(345);
   linkedList.add(node3);
   results[1] = linkedList.size();
   return results;
})();
```

Example 5.2 – Executor format

5.1.1 Recursive construction

The tests are built using recursion, in the sense that when a method call or a constructor call (a declaration) is added to a candidate test, any parameters of that call are declared beforehand. The recursion stops when a constructor call has no parameters *e.g.* in the case of primitives. For the sake of conciseness primitives are in fact inlined as can be seen in the example 5.1. So, in the example 5.1, the

`LinkedList.add` calls each prompt the declaration of a `Node` object, which in turn prompts an inline declaration of a number. The tests *always* start with a declaration of the CUT, and therefore with any declarations that are needed for its constructor's parameters.

The number of method calls made in an attempt to modify the state of the CUT instance is chosen *at random*, with a user-configurable maximum. As we will see in the experimental evaluation, this maximum with regard to the length of method call sequences is significant: it has an effect on code coverage (as well as on test readability, which may or may not be a concern). However, there must be at least one occurrence of the MUT in the *overall* method call sequence, since the sole purpose of the test is to evaluate that method. Note that if there is only one MUT call, it will be at the end, as there is no use in calling more methods *after* we have finished evaluating the MUT.

We initially made the mistake of making only one MUT call in total, but later realised that this could preclude certain portions of the code from being covered. For example, if `LinkedList.add` is the MUT, it needs to be called at least twice for a test to exercise full coverage of that method, as the branches taken in the MUT depend on whether the linked list is empty or not.

Aside from its recursive construction, another feature of the tests' structure is the pooling of parameters.

5.1.2 Pooling parameters

In order to reach full coverage, it is important to enable methods in the tests' method call sequences to manipulate references to *the same object*. Let us take a look at the following code:

```
var linkedList = new LinkedList();

var node1 = new Node(123);
linkedList.add(node1);
var node2 = new Node(234);
linkedList.remove(node2);
```

Example 5.3 – Unreachable code

In that example, a substantial portion of the `LinkedList.remove` method will never be reached: the portion which expects a reference *already contained* in the list. Unless we allow the `LinkedList.remove` method to use parameters defined previously, without necessarily redefining its own, it is impossible to hand it a reference which exists in the list. In other words, it has to be possible to generate:

```
var linkedList = new LinkedList();

var node1 = new Node(123);
linkedList.add(node1);
linkedList.remove(node1);
```

Example 5.4 – Pooling

This is implemented with a 'pooling' mechanism. For every type, there is a pool to which objects are added to when they are declared. This enables any subsequent constructor calls or method calls which need a parameter of that type, to reuse a variable which is already declared in the test. Upon a series of experiments, it was determined that a 25% reuse rate was suitable. The design decision of not making the reuse rate user-configurable was made in order not to overcomplicate Flycatcher's interface.

Note that although tests which are destined to be run in Flycatcher's custom runtime environment do not have the same format as the ones output to the test suite, they *do* share the same underlying structure which has been described in this section. Although we have discussed how tests are constructed, we have purposely ignored one important detail: *we do not know the types of any of the parameters*. How do we know that the `LinkedList.add` method takes an object of type `Node`?

5.2 Types

In generating a test like example 5.1, the type of the variable `linkedList` is known from the start — it is an instance of the CUT. However, initially, the types of the parameters for the `LinkedList` and `Node` constructors are not. For that reason, a special type of object was devised: the `Unknown` type.

5.2.1 `Unknown` type

Early on in the test generation process, this object will replace any parameter which we need but do not know the type of. Inside the Executor, these objects enable us to *collect information* about the parameter that they stand for — we will elaborate on *how* this is done in the next chapter. In most cases, thanks to this information, the `Unknown` objects are eventually substituted with objects of the appropriate type. However, this is what a test might initially look like:

```
var linkedList = new LinkedList();

var unknown1 = new Unknown();
linkedList.add(unknown1);

var unknown2 = new Unknown();
linkedList.add(unknown2);
linkedList.size();

var unknown3 = new Unknown();
linkedList.add(unknown3);
linkedList.size();
```

Example 5.5 – Unknown parameter types

It is worth noting that tests that contain `Unknown` objects are *only* destined for the Executor, they are not suitable as final output, regardless of whether they achieve coverage or not (that coverage is in effect meaningless). We will refer to them as invalid tests.

Eventually, after enough information has been collected to infer that the type of the parameter to `LinkedList.add` is `Node`, the test would look like:

```
var linkedList = new LinkedList();

var unknown1 = new Unknown();
var node1 = new Node(unknown1);
linkedList.add(node1);

var unknown2 = new Unknown();
var node2 = new Node(unknown2);
linkedList.add(node2);
linkedList.size();

var unknown3 = new Unknown();
var node3 = new Node(unknown3);
linkedList.add(node3);
linkedList.size();
```

Example 5.6 – After type inference for `LinkedList.add`

Similarly, when there is enough information to infer that the type of the parameter of the `Node` constructor is a primitive number, we get the test in example 5.2.

5.2.2 Type inference

The type information that is collected when tests are run by the Executor is stored in the `ProgramInfo` object introduced in chapter 4. Upon each Test Generator/Executor iteration for a test, we update `ProgramInfo` using the type

information gathered in the latest Executor run. Although we will explain exactly how this information is collected when we describe the Executor, here we describe how that information is used to infer types for the various test parameters.

Member accesses

For each parameter, the `ProgramInfo` object keeps track of any attempt to access one of its properties. In effect, this records attempts to retrieve *fields* and *methods* from that parameter. By cross-referencing those accesses with the fields and methods stored in `ProgramInfo`'s class definitions, we may be able to deduce the type of that parameter. If more than one type has fields or methods that match the member accesses of the parameter, the one with the highest correspondence is elected.

During the `ProgramInfo` type updates, the following questions are asked to try and determine the type of parameters, in that order:

1. *Does the parameter have any user-defined member accesses?*
   ⇒ If so select the highest match.

2. *Otherwise does the parameter have any member accesses corresponding to the* `Number`[1] *type e.g.* `toExponential`*?*
   ⇒ If so select the *number* primitive type.

3. *Otherwise does the parameter have any member accesses corresponding to the* `String`[2] *type e.g.* `charAt`*?*
   ⇒ If so select the *string* primitive type.

If none of these are true but there are member accesses[3] which do not match any known type, this means that we are faced with a non-standard or non-supported class which is not accessible in the scope of the program. In that case we abort the test generation process for that particular method with a warning.

If however there are no member accesses, the parameter may be a primitive, in which case we look at its 'primitive score' accumulated during Executor runs. Note that the question of whether the parameter has member accesses comes first, as it can rule out the possibility that the parameter is a primitive, but not vice-versa.

Primitive scoring

The primitive score of a parameter is an object which accumulates the likelihood of a parameter being of a primitive type based on the operations it is involved

[1] the object counterpart of the *number* primitive type
[2] the object counterpart of the *string* primitive type
[3] except for the hidden properties inherited from `Object` in JavaScript which are discounted

in. Like the member accesses, this information is recorded when a candidate test is run in the Executor. The primitive types taken into account are *number* and *string*. We do not deal with *null*, *undefined* and *boolean* as there are no strong hints that can be used to infer any of these types[4].

The primitive scoring object thus corresponds to:

```
{
   "number" : 15,
   "string" : 12
}
```

Example 5.7 – Primitive scoring

When type inference takes place for a parameter that is suspected of being a primitive, the most likely primitive type is chosen based on the scores in the accumulator object.

Delaying type inference

Because we do not want to jump to conclusions too quickly when inferring types using the member accesses and primitive scores, we do not start updating the `ProgramInfo` object until a sufficient number of constructor or method calls *involving* that parameter have been made (during the Test Generator/Executor iterations). For example in the following code, the parameter of the `LinkedList.add` method is involved or 'used' in *two* calls.

```
var linkedList = new LinkedList();

var unknown1 = new Unknown();
linkedList.add(unknown1); // first 'use' of add's parameter

var unknown2 = new Unknown();
linkedList.add(unknown2); // second 'use' of add's parameter
linkedList.size();
```

Example 5.8 – Parameter uses

This delay, characterised as a minimum number of parameter uses, is user-configurable, as an optimal value for it is highly dependent on the program under test. For example, if a user *knows* that a parameter is only active in a path that is executed infrequently, they may set a high value for the delay before type inference. By making sure that a parameter has had an *adequate* number of opportunities to collect type information, the user can ensure that type inference does not take place too early, when it has a higher chance of being inaccurate. On the other hand, if the user knows that a parameter is active *every time* its

[4] the loss in terms of coverage is negligible as in most cases these values produce the same behaviour as `0` or `1`

function gets called *i.e.* it lies on every path inside that function, the user may want to give the 'minimum number of uses' variable a low value to save time.

At first it seemed that a more telling and intuitive measure of confidence about inferring a type for a parameter was how many member accesses had been recorded, or how good the best primitive score was for it. However, any lower limit on such variables makes *too strong assumptions* about the program under test. For example, in a scenario where it is decided that three member accesses provide enough information to make a confident type choice: what if a class only has one or two members?

When the delay expires, if the type inference is inconclusive due to the absence of member accesses and null primitive scores, this is a sign that the parameter is seldom or never accessed. However, this parameter will remain an `Unknown` in the Executor and Flycatcher will not be able to terminate if we do not give it a real type. Hence, we make the leap of faith that, given that the user-configurable delay has expired, the parameter is *not* used and we replace it with a random substitute primitive value, warning the user. An illustration of this scenario is the linked list in Appendix A.1, as the implementation itself does not make use of the data in the nodes. This is a common pattern which needs to be accounted for. In many cases like this one, the substitution works and enables tests to become valid (rid of `Unknowns`).

In the event that the parameter is in fact used in the program but happened not to be in the delay chosen, tests will fail due to type errors and the user can see that in the logs. They may then adjust the delay variable to suit the needs of their program.

5.3 Custom data generators

When a user-defined type is inferred for a parameter, a suitable object can be constructed in tests using the appropriate class definition, available in the `ProgramInfo` object. However, regarding the two primitive types *string* and *number* that we are concerned with, a random value needs to be generated.

Many programs require specific input data and simply generating a number from the space of natural numbers or constructing a string from random combinations of ASCII characters will fail to achieve code coverage in those cases. For example, it is infeasible to try and achieve coverage in a program that validates 13-digit International Standard Book Numbers with randomly generated numbers. Hence, Flycatcher is equipped with an extremely convenient way of specifying custom data generators: with regular expressions. The regular expressions that specify a string or a number generator are defined as optional strings on the command line (see usage in section 3.4) and they have to conform to a JavaScript `RegExp`.

The JavaScript library used to *generate* random matches *from* JavaScript reg-

ular expressions is randexp, available as a Node.js module[5]. Example 5.9 shows a Flycatcher custom string generator that generates 4-character long hexadecimal strings and a custom number generator that generates 13-digit long numbers:

```
// 4-character long hexadecimal string generator
"[A-F0-9]{4}"
// 13-digit long number generator
"\d{13}"
```

Example 5.9 – Custom data generators

Note that the regular expression that specifies the number generator should only match representations that are valid JavaScript numbers, albeit in string format. The generated string is coerced to a number, therefore if a representation contains invalid characters, it will cause a `NaN` to be generated.

5.4 Outcome

5.4.1 Output

The role of the Test Generator is to generate *candidate* tests, but the end goal of Flycatcher is to generate a suite of unit tests. So how is a candidate test elected to appear in the unit test suite? A candidate test is a useful unit test if it fulfils the following criteria:

- It achieved *new* coverage in the Executor (the current coverage is that achieved by other tests so far)
- It is valid *i.e.* it contains no `Unknowns`

If a test fulfils those two criteria it is added to the suite of unit tests, which is output in a chosen format among a choice of JavaScript unit testing frameworks. The current unit testing formats available are node-unit, expresso and Node.js's assert module. More framework formats may be available in the future. If none of these frameworks suit the user, the unit tests are output by default in simple JavaScript code with assertions. The unit test suite output is meant to be a regression test suite. That is to say, all the tests in it pass when run with the current program under test. They are useful to uncover errors if the program under test changes and *breaks* any of the unit tests.

However, valid tests that failed inside the Executor due to an exception being thrown, are added to a log file for the user's attention. If the tests failed due to bugs or a lack of defensive programming, the user can use this feedback to fix these defects. If the tests failed because of an incorrect type inference, the user can adjust the delay before type inference, such that a better, more confident estimate is made.

[5] `"https://github.com/benburkert/randexp"`

5.4.2 Timeouts

A test may contain no `Unknowns` but still fail to ever achieve any new coverage, for example if a portion of the code is simply not reachable in the program. In this case, Flycatcher may loop as it may never achieve full coverage of a MUT. Hence, there are two timeouts in place to handle this scenario:

1. a strict timeout in seconds
2. a timeout characterised by aborting after a specific number of test runs during which coverage did not improve

Both timeouts are user-configurable variables, specified by optional command line parameters (see usage in section 3.4). Another reason than dead code for which new coverage might be unattainable is simply bugs in the program under test. Potential bugs may preclude any new coverage from being achieved because of thrown exceptions. The two timeouts also apply in this case, and the failing tests can be found in the logs. Note that two types of errors do not fit into this category: *stack overflow* and *infinite loops*. Due to the nature of these errors, the former crashes Flycatcher and the latter causes it to hang. Finally, if termination is not caused by timeouts, then Flycatcher stops generating tests for a MUT when it achieves its full coverage.

The next chapter is devoted to shedding light on the part of the implementation that has been left out so far: the responsibilities and inner workings of the custom execution environment or Executor.

CHAPTER 6

Developing a custom runtime environment

The tests that are generated by the Test Generator need to be executed by Flycatcher for two reasons:

1. To collect runtime information about parameter types, such that tests with *accurate* types can be generated — only these *valid* tests can be output and serve as unit tests
2. To evaluate the code coverage achieved by valid tests, in order to elect the ones that make a difference

To those ends, a custom runtime environment was created: the Executor. The Executor uses the same vm package used by the Analyser in order to create the runtime environment — it just passes to it a sandbox with all of the elements that it wants to make available. In this chapter, we elaborate on how, in that environment, runtime type information is collected and code coverage is tracked for the MUT.

6.1 Collecting type information

Collecting information at runtime about the types of parameters for method calls and constructor calls in candidate tests involves the `Unknown` type which was introduced in the previous chapter. However, for clarity, we simplified the declaration of `Unknown` objects in candidate tests as:

```
var unknown = new Unknown();
```

In fact, these special objects that are capable of collecting type information at runtime require the following declaration instead, which we will elaborate on:

```
var unknown = __proxy__(className, functionName, paramIndex);
```

The __proxy__ method is a special method[1], which creates a Proxy object tailored for type information collection, that we shall name `CollectorProxy` to be precise. The information is collected inside the `ProgramInfo` object, as discussed in chapter 5. Hence, the `CollectorProxy`'s handler needs to be instantiated with information that lets it access the part of `ProgramInfo` corresponding to the parameter it stands for:

- `className`: the parameter that the proxy stands for belongs to a function, that function belongs to a class — this is the name of that class
- `functionName`: the parameter that the proxy stands for belongs to a function, this is the name of that function (which is identical to `className` in the case of a constructor)
- `paramIndex`: the parameter that the proxy stands for belongs to a function, this is the index of that parameter among the function's parameters

The type information collected at runtime takes two forms:

1. Recording member accesses
2. Accumulating a score for primitives

6.1.1 Recording member accesses

The member accesses are recorded using the `get` trap of the `CollectorProxy`: the trap translates to `handler.get(receiver,name)` where name is the name of the property that was accessed. And we know from the initialisation of the `CollectorProxy`'s handler, where in `ProgramInfo` to store that information.

However, when a property is accessed, the `get` must return an appropriate object for execution to carry on. Much like in the Analyser, we return a *Function Proxy* that can respond to any operation, but this is not to be confused with the `CollectorProxy` — the proxy we return no longer stands for a parameter and as such does not collect any type information (its role is simply to avoid crashes). The only difference is that we randomise[2] the primitives returned by that proxy, so as to diversify the code exploration achieved by invalid candidate tests. Doing so diversifies our collection of type information in the early stages of the test generation process.

[1] the name __proxy__ should avoid name clashes with classes in the program under test

[2] 0 is returned more often as it is often a significant value in path exploration

6.1.2 Accumulating primitive scores

The operators in table 6.1, when applied to operands, translate at a lower level into the internal function `[[getValue]]` being called on the operands. This function in turn calls the `valueOf` method on the members if they are objects, which instances of the `CollectorProxy` are. This enables the `CollectorProxys` to trap the requests to get the `valueOf` method, as they would any other method. Note that the operators: `&&`, `||`, `!`, `=`, `===` and `!==` are missing from the table as they do not yield a `valueOf` call and thus cannot be trapped.

Arithmetic operators	`+, -, *, /, %, ++, --, unary -, unary +`
Assignment Operators	`*=, /=, %=, +=, -=, ≪=, ≫=, ⋙=, &=, ^=, |=`
Bitwise Operators	`&, |, ^, ~, ≪, ≫, ⋙`
Comparison Operators	`==, !=, >, >=, <, <=`
String Operators	`+, +=`

Table 6.1 – Operators yielding `valueOf`

When the `valueOf` call is trapped, the handler only needs to compare the `name` in `handler.get(receiver,name)` to `valueOf` to determine whether it is dealing with a primitive operation. However, the problem is that `valueOf` does not teach us anything about what sort of primitive we are dealing with and we cannot use it to calculate scores for primitive types. This led us to develop the following steps in order to deduce a primitive score from a primitive operation:

1. Determine if the `get` trap corresponds to a primitive operation *i.e.* if it is a `valueOf` access
2. If it is, throw an exception and *catch it within the handler*
3. In the `catch` body, use the Node.js `stack-trace`[3] module to retrieve the line where the primitive operation happened
4. Scan that line of source code for *hints* about the primitive type of the parameter
5. Based on the hints found, increase the primitive scores in the score accumulator object

Table 6.2 shows the hints that are looked out for and the corresponding increases in primitive type scores. The scoring method in the table is not based on any formal heuristics, only on our extensive programming experience in JavaScript as well as experimentation. For example, if applied to a *string*, the `++` operator yields a `TypeError`, hence why the `++` hint confers a particularly high score to the other type, *number*.

[3] `https://github.com/felixge/node-stack-trace`

Hint	*number*	*string*
`++`, `--`	10	0
`>`, `<`	2	1
`-`, `*`, `/`, `%`, `≪`, `≫`, `⋙`, `^`, `|`, `&`, `~`	2	0
[0 - 9]$^{+}$	5	0
double quotes[4]	0	5

Table 6.2 – Primitive scoring

Other operators or hints have been purposely omitted, since they do not tip the balance in any particular direction. For example, the + operator is used with strings as much as it is used with numbers, and thus does not constitute a helpful hint.

Step 4 of the above algorithm is carried out using regular expression matching. However, when searching for the hints in table 6.2, one must be careful not to match prefixes of other operators. For example, if the symbol found is a `--` one must be careful not to match a single `-`. The problem is that although the JavaScript RegExp syntax allows for *negative lookahead* (matching unless the match is followed by something), it does not allow for *negative lookbehind* (matching unless the match is preceded by something). Thankfully, the XRegExp[5] library compensates for that and provides the missing functionality.

In summary, the resulting primitive scores correspond, like the collection of member accesses, to information concerning the type of a particular parameter. This information is stored in the `ProgramInfo` object, and the Test Generator uses it to try and infer a type for that parameter.

6.1.3 Trap threshold

A certain issue was uncovered with the trapping mechanism devised for type collection: it could lead to the infinite execution of terminating programs. This was discovered with a JavaScript implementation of the Fibonacci algorithm:

```
Fibonacci.prototype.compute = function(n) {
    if (n>1) return this.compute(n-1) + this.compute(n-2);
    else return n;
}
```

Example 6.1 – Fibonacci in JavaScript

[4] single quotes can be ignored, as if they appear in the context of the Executor it means that they harbour a double quote, which will be found

[5] `"http://xregexp.com/"`

The problem is that when the > and - operators are called, they are trapped, and the value returned is not the result of the operation but a random value. Hence, the induction that makes the Fibonacci algorithm terminate is broken and a program which is supposed to terminate ends up hanging.

As a solution to this, a trap 'threshold' was implemented, which only allows the `CollectorProxy` to trap 15 times. When the trap threshold is exceeded, the whole test is discarded. Given that trapping results in the collection of type information, upon a certain number of iterations, the parameter will change from an `Unknown` to a resolved type, and the infinite execution will have been avoided. The value of the trap threshold is not significant — what matters is that it prevents infinite execution when the looping is due to the `Unknown` objects' behaviour.

Having discussed the collection of type information and its caveats, we now move on to discussing the second purpose of the Executor: tracking code coverage.

6.2 Code coverage

When doing *structural testing i.e.* testing the internal workings of a program, we are interested in executing as many paths as possible in that program, regardless of whether they are likely to be used in practice. The quality measure of a test in the context of structural testing, also known as *white-box testing*, is therefore *code coverage*, which reflects how error-free a program is. In practice, generating tests to cover every single possible execution path in a program is infeasible. Hence, in Flycatcher, we resort to a weaker measure of coverage: *statement coverage*, which tracks the statements that are executed.

In Flycatcher the target of the output unit testing suite is the CUT, as by default all of its methods are tested one by one. But as these methods each become the MUT, it is *their* individual coverage that we are interested in. In other words, the candidate tests generated are geared towards a particular method each time, and it is the coverage achieved *in that method* that can give us the quality of a test. Coverage in some other remote part of the program is not relevant to how well we are testing a particular method. Hence, code coverage needs to be tracked and reported for each MUT independently.

In order to do so, the following steps are necessary:

1. When a MUT is selected, *all the statements* in the definition of the MUT used by the Executor must be wrapped with a callback that updates coverage for that method
2. The Executor environment must implement that callback

6.2.1 Wrapping the MUT statements

Wrapping the MUT statements is done with the help of another Node.js package: burrito[6]. Wrapping with that package is done like so:

```
var burrito = require("burrito");

var wrapped = burrito("foo()", function (node) {
    node.wrap("callback(%s)");
});

console.log(wrapped); // prints callback(foo());
```

Example 6.2 – Wrapping with burrito

However, in our case we are interested in attributing indices to statements such that when that portion of code executes, the callback registers that the statement with that index has been covered. The wrapping statement thus looks like this instead:

```
node.wrap("callback(" + index + "); %s;");
// where index is incremented when a new node is wrapped
```

Example 6.3 – Wrapping in Flycatcher

However, the wrapping varies for different kinds of statements, as the calls to the callback have to be inserted in a way that they do not break JavaScript syntax and crash the program. For instance, in example 6.4, the coverage callback called with index 2 pertains to the `return` statement and differs in format from the callbacks with indices 4 and 6 used to wrap the `compute` method calls.

```
Fibonacci.prototype.compute = function(n) {
  if (__coverage__(1)(n > 1)) {
    __coverage__(2);
    return __coverage__(3)(
               __coverage__(4)(this.compute(__coverage__(5)(n - 1)))
                    +
               __coverage__(6)(this.compute(__coverage__(7)(n - 2)))
           );
  } else {
    __coverage__(8);
    return n;
  }
};
```

Example 6.4 – Wrapped Fibonacci

Hence, to track coverage in the Executor it was necessary to understand and adapt the wrapping process carried out by burrito. That wrapping process hap-

[6] `https://github.com/substack/node-burrito`

pens in three steps:

1. The code is parsed into an AST using the parser from the uglify-js[7] module
2. A wrapper function is mapped onto every node in the AST using the traverse[8] module
3. The transformed AST which contains the wrapped statements is rendered into code

Therefore, understanding the wrapping process involved understanding and manipulating the AST, and wrapping its nodes appropriately. This was challenging because the AST in question is very poorly documented[9], and working with it was thus laborious. Nevertheless, due to its portability, we believe that using burrito for our instrumentation purposes was the best solution available. Indeed, the other coverage tools available for JavaScript were deeply embedded in a testing application and difficultly reusable by Flycatcher.

At the end of the instrumentation process, the list of indices used by the coverage callback serves as a representation for a MUT's statements. That list can be used to keep track of coverage inside the MUT.

6.2.2 Implementing the callback

The Executor must implement the coverage callback which is called when a statement is executed. This callback manipulates the list of indices gathered during the instrumentation process, updating it with the fact that a certain statement has been reached. In other words, it sets the value of the index it is called with to `true`. For example 6.4, the list would initially be:

```
[false, false, false, false, false, false, false, false]
```

After the recursive case is executed it would be:

```
[true, true, true, true, true, true, true, false]
```

And finally when the base case is executed:

```
[true, true, true, true, true, true, true, true]
```

As a result of this coverage mechanism, the Executor can report:

[7] `"https://github.com/mishoo/UglifyJS"`

[8] `"https://github.com/substack/js-traverse"`

[9] the only pseudo-specification available is: `"http://marijnhaverbeke.nl/parse-js/as.txt"`

- Whether a test achieves any *new* coverage for a MUT, and is therefore a useful test to be output in the unit test suite
- What the current total code coverage for a MUT is, which is used to inform the user of coverage progress, and terminate upon full coverage

CHAPTER 7

Evaluation

In this chapter, we evaluate Flycatcher quantitatively by running Flycatcher with a series of benchmark programs and observing the results. In this experimental evaluation we are interested in examining some of Flycatcher's features as well as assessing its overall success in terms of code coverage. We start by introducing the choice of benchmark programs, before presenting the experiments and finally discussing them.

7.1 Choosing the benchmark suite

In choosing a suite of programs for evaluating Flycatcher we had the following objectives in mind:

- Demonstrating Flycatcher's ability to infer primitive types
- Demonstrating Flycatcher's ability to infer user-defined types
- Demonstrating that Flycatcher works with methods of various size, complexity and arity
- Demonstrating Flycatcher's ability to achieve high coverage for various methods in a reasonable time

However the constraints imposed to us by the current limitations of the Flycatcher application, made finding benchmark tests difficult. The constraints are the following:

- Flycatcher does not handle the inference of *Array* parameters
- Flycatcher does not handle the inference of *Function* parameters

The reason why these constraints made it difficult to find benchmark programs is that array parameters are commonplace and because functions are first-class objects in JavaScript, they frequently occur as parameters too. This also indirectly imposed a restriction on the size of the benchmark programs — most significant libraries and modules make use of either arrays or functions as parameters at some point. This is also why established JavaScript benchmark suites such as SunSpider or the V8 benchmark suite couldn't be used.

With these objectives and constraints in mind, the list of methods in table 7.1 was put together. Some of the programs are custom, many were found using Node.js's open-source module registry npm and others were taken from the V8 performance benchmark suite.

7.1.1 Triangle types

The `Triangle` example is used in many testing papers for automatic test generation. It takes three numerical inputs and determines whether they can form a triangle. If so, it returns the type of the triangle *i.e.* whether it is equilateral, isoceles or scalene. This example was chosen because it demonstrates Flycatcher's ability to narrow down the search space for good test programs, by using the same parameters more than once inside the generated tests. Without that ability, generating three numerical inputs out of the set of natural numbers so that they form an equilateral triangle would be extremely inefficient. No custom data generators with this class in the experiments.

7.1.2 Doubly circular linked list

The doubly circular linked list example was picked because it demonstrates Flycatcher's ability to infer user-defined types as well as deal with parameters which are not used in the program under test itself. The circularity of this data structure also shows that Flycatcher handles scenarios where termination issues might arise in the context of test generation. The implementation used is from computer-science-in-javascript[1].

7.1.3 Binary trees

A variety of binary trees were chosen as benchmarks because they present interesting control structures for code coverage, as well as the requirement that they infer a user-defined type: the type for the trees' nodes. The standard `BinarySearchTree` implementation is from computer-science-in-javascript[2]. The `RedBlackTree` (a self-adjusting BST) implementation is from red-black-tree-js[2]. The `SplayTree` (a self-adjusting BST with quick retrieval of recently accessed nodes) implementation is part of the Google V8 benchmark suite[3]. No custom

[1] "https://github.com/nzakas/computer-science-in-javascript"

[2] "https://github.com/jeffreyolchovy/red-black-tree-js"

[3] "https://github.com/hakobera/node-v8-benchmark-suite"

data generators were used with these programs.

7.1.4 Luhn Algorithm

The Luhn Algorithm is a checksum algorithm used to validate the format of credit card numbers. This example demonstrates the use of custom data generators in order to narrow down the search space for code coverage. We specify the following `RegExp` as the random string generator, which represents a string of digits which may contain a character as well (to also test with `NaN`): `[0-9]+a?`. The code can be found in computer-science-in-javascript[2].

7.1.5 Base 64

The `Base64` class simply encodes and decodes text strings to and from a radix-64 representation. This program was chosen to test the efficiency of code coverage of its control structures. Custom data generators were used to achieve full coverage, as non-ASCII parameters take a specific path in the *encode* method, as do non-base-64 strings in the *decode* method. The custom string generators are, therefore respectively, `\w+\u0100?` and `\w+`.

7.1.6 SHA1

The `SHA1` algorithm generates a SHA-1 secure hash of a string. The implementation is taken from Chris Veness[4].

7.1.7 Poker

The benchmark method with the most deeply nested structure is the `Poker` class's *rankHand* method. It carries out hundreds of comparisons and calculations in order to return the absolute rank of a hand of cards in Texas Hold'em poker. The input thus has to conform to a hand of poker cards, which is defined in the program as a string of five characters. A custom string generator is thus used for that purpose, with the following regular expression: `[AKQJT98765432]{5}`. The code is from node-poker[5].

7.2 Experiments

All the experiments are carried out on Mac OS X v10.6 with a 2.4GHz Intel Core 2 Duo processor and 4GB of RAM. Any of the configurable parameters are either specified or their default value is used. The noteworthy default values are:

- type inference delay: 20

[4] "http://www.movable-type.co.uk/scripts/sha1.html"

[5] "https://github.com/mjhbell/node-poker"

- maximum sequence length: 10
- number generator: `\d{10,20}`
- string generator: `\w{10,20}`

7.2.1 Effect of varying the type inference delay

In this first experiment we analyse the effect of varying the type inference delay discussed in section 5.2.2 (see usage in section 3.2). Let us recall that this delay is put in place in order for Flycatcher to make confident type inferences, based on enough data. If there is not enough data when type inference is attempted for a parameter, we consider that the type inference is *unsuccessful*. In this trial, we observe the relation between the type inference delay and unsuccessful type inferences for the methods from the benchmarking set, using an average over twenty runs and displaying the results for two of the methods in figures 7.1 and 7.2.

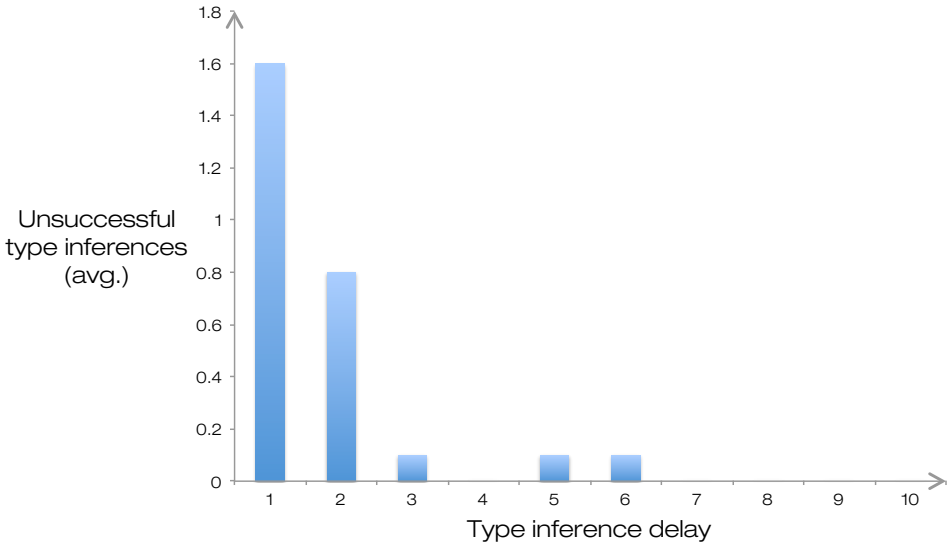

Figure 7.1 – Results for method `Triangle.getType`

7.2.2 Effect of varying the maximum length of tests

In the second experiment, we vary the user-configurable variable that sets the length of method call sequences in tests (see usage in section 3.2). We observe what the effect of modifying this variable is for all of the methods in the benchmark set. To give us a more comprehensive set of results to analyse, this process is carried out three times with varying 'strict' timeouts of: 1 second, 5 seconds

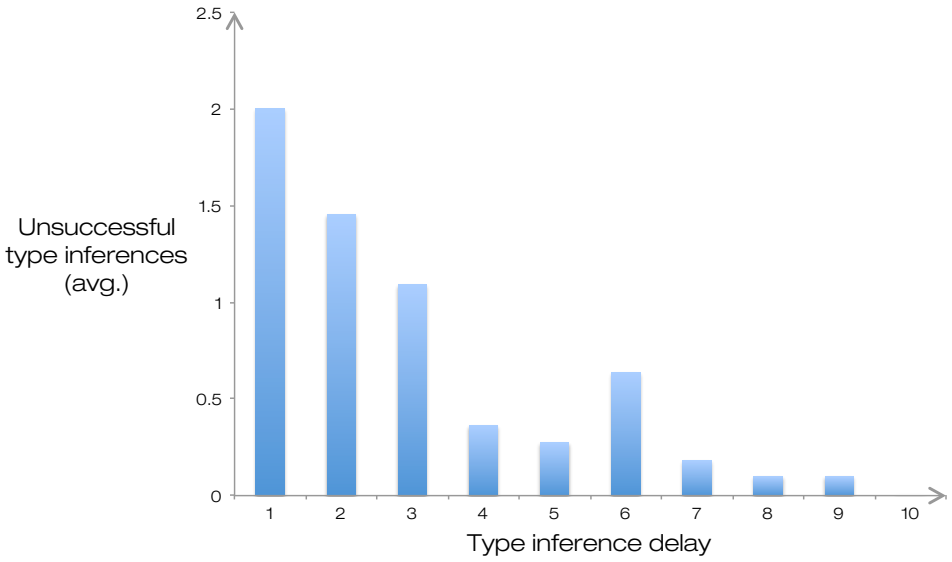

Figure 7.2 – Results for method `BinarySearchTree.add`

and 20 seconds. The results appear in figures 7.3, 7.4 and 7.5 and each entry constitutes an average over 20 runs.

7.3 Discussion

7.3.1 Effect of varying the type inference delay

From the first experiment, we can observe that there is a clear relation between unsuccessful type inferences and type inference delay[6]: the smaller the type inference delay, the higher the number of unsuccessful type inferences. Hence, this user-configurable parameter which reflects the confidence of the type inference estimates *does* effect test generation results and thus deserves careful consideration.

As we can see from the results, the number of unsuccessful type inferences tends to zero as the type inference delay increases. This led to a choice of 20 for the default value of this variable within the application, such that by default, highly confident type estimates are made by the test generator. Although the results are displayed for only 2 of the 22 benchmark methods for conciseness, the experiment was carried out for all of the methods, with equivalent results.

[6]to reiterate, the type inference delay variable is expressed in terms of the minimum number of uses of a parameter before type inference

7.3.2 Effect of varying the maximum length of tests

The second experiment teaches us that the length of method call sequences in generated tests also has a significant impact on test generation. First of all, in some cases we observe that if the maximum length of tests is too small, certain parts of a MUT may be *unreachable*. For example, when the method sequence length is 1, the code coverage for method `LinkedList.append` stays stuck around 60%, no matter how long the timeout.

Next, we also observe that although one may initially think that longer tests are synonymous of more efficient code coverage, the results in this experiment proves this wrong. Due to the impact of generating and running very large tests, it is *not* always the case that bigger tests yield better coverage. In fact we can observe a trend among the benchmark methods whereby the code coverage peaks for lengths between 5 and 20 but drops afterwards. This is particularly observable in figure 7.3, for methods `LinkedList.remove`, `LinkedList.insertAfter` or `SlapyTree.insert`.

Finally, the effect of modifying the maximum length of tests is not always predictable as can be seen in figure 7.3 with the `SplayTree` methods `splay`, `remove` and `findMax`. In those cases, increasing the maximum length of method call sequences has a varying effect on coverage efficiency. It is not exactly clear what the best approach is to get the most efficient code coverage.

All of the above observations point to the same fact: finding the optimal length of tests for efficient coverage of a MUT is not only *necessary* for performance, but its unpredictability means that it is no task for humans. This leads us to one of the clear limitations of Flycatcher: the randomness of the test generation process, particularly when it comes to the length of tests, is restrictive.

7.3.3 Code coverage

Experiment in section 7.2.2, on top of showing the effect of the length of tests, revealed Flycatcher's success in terms of overall code coverage. Table 7.2 shows a summary of the proportion of methods for which full coverage is achieved. The results are highly promising: only 2 out of the 22 methods are not fully covered in under 20 seconds and 100% coverage is reached for 14 of them in under a second.

In structural testing, the major indicator of the quality of a test is the amount of code coverage achieved, which by extension is also a sound indicator of the quality of a test generation tool. It would thus be fair to say that full coverage for 91% of the methods in a benchmark is evidence of a successful test generation tool.

Although it is written in and for Ruby, the tool most comparable to Flycatcher is RuTeG [26]. Given that the target languages for each tool are different and that JavaScript is deemed 'faster' than Ruby [13], it seems unfair to pit the tools against each other. Nevertheless, despite our unorthodox benchmark suite,

two of the methods we used are very similar to methods used in the evaluation of RUTEG, namely:

- `Triangle.triangle_type` (RUTEG) — `Triangle.getType` (Flycatcher)
- `RBTree.rb_insert` (RUTEG) — `RedBlackTree.insert` (Flycatcher)

For both of these methods, Flycatcher performs better in terms of the code coverage achieved in under 20 seconds:

- `Triangle.getType`: 100% coverage versus approx. 87%
- `RedBlackTree.insert`: 100% coverage versus approx. 70%

Once more however, this comparison is nothing more than an indicator that our tool is roughly on a par with comparable state-of-the-art tools in other languages — it has no implications with respect to the relative quality of each tool, given their differences.

7.3.4 Limitations

Despite producing satisfying results with the chosen benchmark suite, Flycatcher presents certain limitations. A major design decision behind Flycatcher was to make the tool as autonomous and convenient as possible. To a certain extent, this has been achieved, as the minimal input a tester has to provide to get a suite of unit tests for an entire class is:

- the name of the file with the class
- the name of the class

In the best case scenario, providing only those two arguments will supply the tester with a full suite of unit tests for the class as well as a log of the tests that failed. However, this level of autonomy does have its drawbacks: by entrusting decisions such as the optimal length of tests or the seed for data generators to Flycatcher, the tester might lose on efficiency or accuracy. The high overall accuracy and efficiency of the results in the experiments in this chapter owe in part to the explicit specification of the maximum length of tests and custom data generators. For example, the method `Poker.rankHand` cannot achieve *any* coverage under 20 seconds by using the default string generator, as it expects a particular string format — the odds are simply too low. However, because we *do* value autonomy in such a tool, as we know how much programmers prefer programming to testing, we feel that the solution is not to ask for more input from the user, but to make Flycatcher better. In other words, Flycatcher is restricted by its use of random test generation, be it for data itself or the test cases. And although this can be partly mitigated with input from the user, as in the experiments, in order to be meaningfully autonomous without compromising accuracy or efficiency,

Flycatcher needs to become more 'clever' in its exploration of the search space of possible tests.

Another major limitation of Flycatcher is the lack of support for the standard objects *Function* and *Array*. Arrays are commonplace function parameters in programs and in JavaScript, so are function objects. As a result, not handling these constructs imposes strong limitations on the programs that can be tested by Flycatcher. This was evident in the benchmark suite selection stage, where we were precluded by this limitation from using any of the established benchmark suites or substantial JavaScript libraries. This in turn represents a threat to the validity of our results as we face a selection bias — the lack of support for *Array* and *Function* impacts the size of programs that can be tested. Because they have such implications, we feel that it is worth outlining why supporting these types is a challenge.

Functions: When automatically generating tests, all we have to begin with is the program under test. With regard to a function parameter, this is what can *possibly* be learnt from the program:

1. finding out that it is a function (initially)
2. what types its parameters belong to
3. what type its return value belongs to

However, as far as we understand, it is not possible to determine what the function in question is or does from this information.

Arrays: Arrays are known to pose problems in automatic test data generation [40]. In Flycatcher, the fact that we use proxies which return random values when called with a primitive operator, means that we cannot have 'faith' in any of the primitives inside the program while generating tests. This makes it difficult to discover the length of arrays. Moreover, because arrays allow for heterogenous types in JavaScript, every entry has to be inferred and generated separately, which adds to the challenge and is likely to be an expensive process.

Flycatcher also does not currently support parameters of the other two standard types `RegExp` and `Date`, but these have less serious implications and their implementation poses less problems.

In summary, Flycatcher is limited by the fact that it uses a random strategy for test generation. Despite that this can be alleviated by input from the user, the fact remains that the application would benefit from using search-based heuristics to guide the exploration for test cases. Moreover, the lack of support for certain crucial types of programming constructs, albeit for good reasons, means that Flycatcher can only be used with a certain class of programs, and thus lacks generality.

Class	Method	LOC
`Triangle`		
	getType	38
`LinkedList`		
	append	15
	remove	16
	prepend	12
	insertAfter	8
	insertBefore	8
	at	6
`BinarySearchTree`		
	add	49
	contains	26
	remove	147
	size	9
`RedBlackTree`		
	insert	22
	contains	14
`SplayTree`		
	splay	60
	insert	23
	remove	23
	findMax	10
`Luhn Algorithm`		
	isValidIdentifer	40
`Base 64`		
	encode	45
	decode	50
`SHA1`		
	hash	72
`Poker`		
	rankHand	437

Table 7.1 – Benchmark methods

Timeout	100% coverage
1s	14/22
5s	20/22
20s	20/22

Table 7.2 – Proportion of methods with full coverage

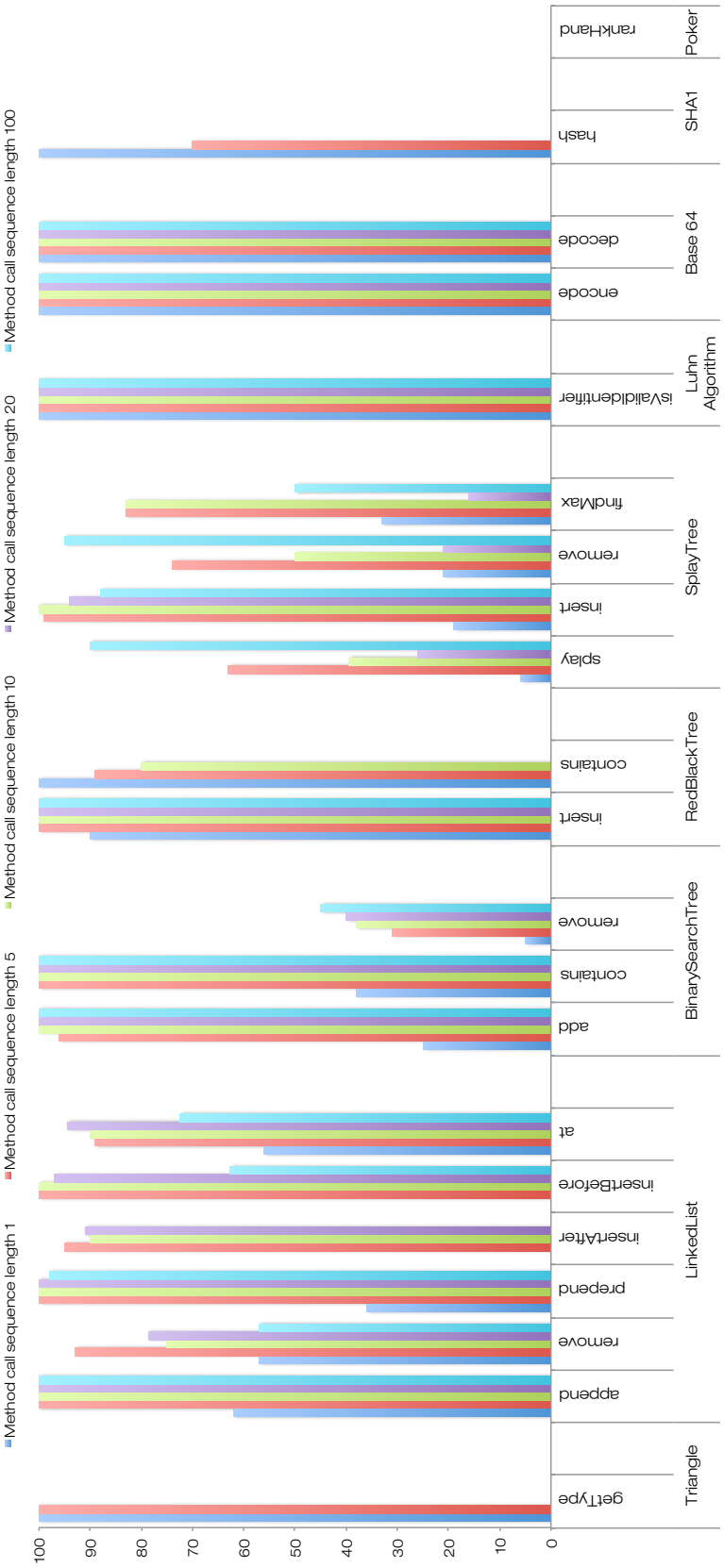

Figure 7.3 – Results with 1s timeout

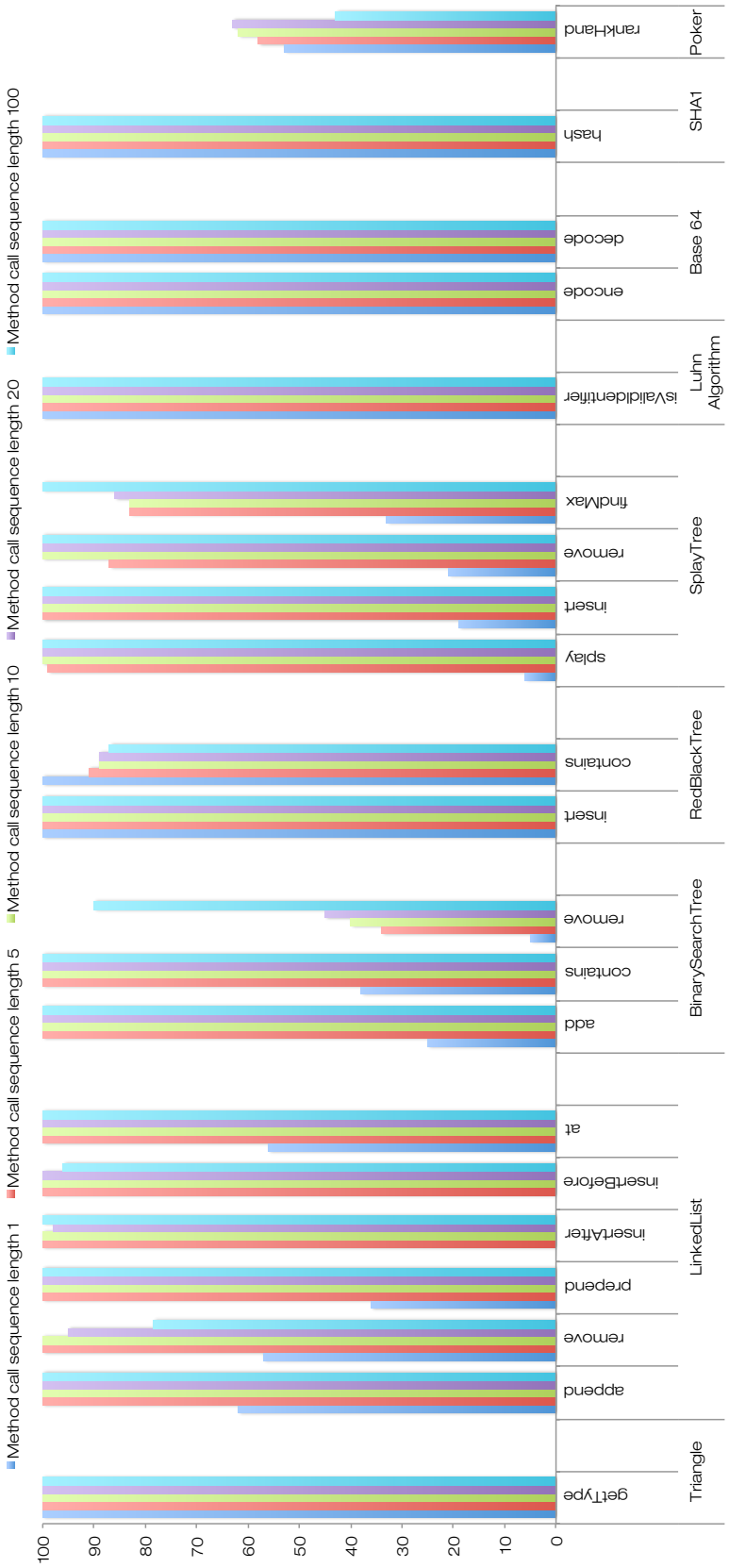

Figure 7.4 – Results with 5s timeout

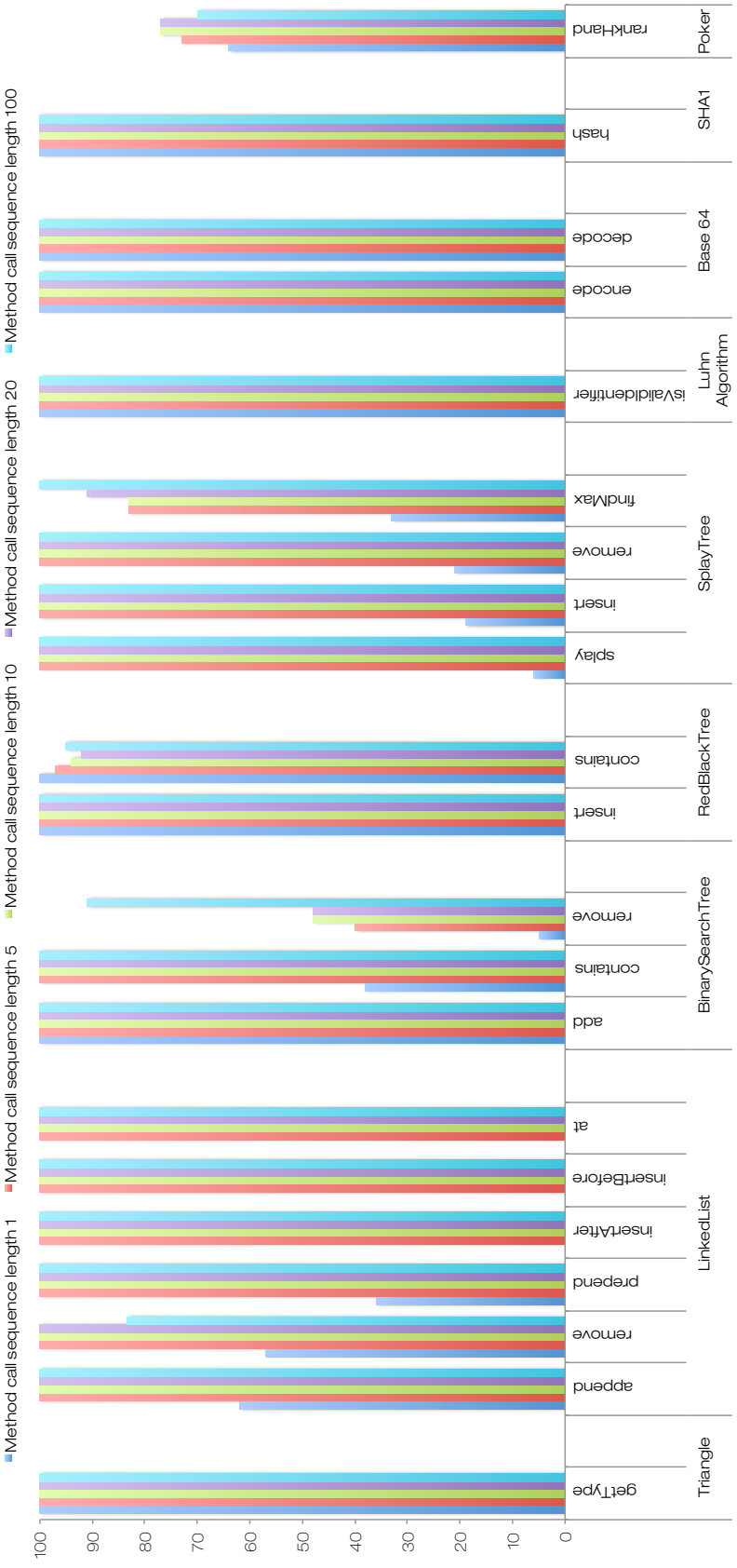

Figure 7.5 – Results with 20s timeout

CHAPTER 8

Conclusion

We have presented Flycatcher, an automatic unit test generation tool for and in JavaScript. Flycatcher is capable of generating accurately-typed tests by inferring the types of parameters based on how these parameters are used within the programs under test. In doing so, Flycatcher overcomes the major obstacle associated with automatic test generation in a dynamic language: discovering parameter types in the absence of static types. Yet, with respect to what types *can* be discovered, Flycatcher presents limitations. User-defined types (including polymorphic ones) as well as primitive types can successfully be inferred by Flycatcher, but there is a hindering lack of support for the JavaScript types *Array* and *Function*. However, this does fit in with our early objective of designing an automatic test generation tool for a comprehensive *subset* of the JavaScript language. With regard to that subset, the high code coverage results achieved by Flycatcher for our 22 benchmark methods are promising and indicative of a valuable automatic test generation tool.

Another essential feature of our application is that it can work autonomously. But this is only the case to a certain extent, as it *can* produce results without the need for user input, but in many cases the absence of this information can drastically impact coverage results. This is due to the fact that the test generation process in Flycatcher is largely *random*, and thus unlikely to exercise low-probability paths in a program. It is therefore fair to say that full autonomy is only possible with Flycatcher in the best scenarios, when the programs under test do not contain improbable paths. In other cases, specifying variables such as custom data generators or an adequate length for tests, is necessary. Despite that, Flycatcher does constitute a considerable gain in autonomy compared to the existing JavaScript test generation tools [38, 2, 19, 5]. Indeed, it is less work to specify a custom data generator for a class using a regular expression on the command line, than to annotate every method with an additional 'contract'. Moreover, unlike the existing work, Flycatcher handles the class-based programming style which is now predominant in JavaScript development.

To conclude, we contribute to the field of automatic test generation by propos-

ing a powerful testing tool and offering new insights into dealing with dynamically-typed languages in that field. Furthermore, we also feel that our project has the potential of being meaningful for the development of the JavaScript language as a whole. Due to the extensive and critical use in Flycatcher of state-of-the-art features being proposed in the latest version of the ECMAScript standard, it seems that our project also acts as a strong endorsement of these features by demonstrating their wide-ranging and powerful applications.

Future Work

Much of the further work that is needed for Flycatcher is aimed towards tackling the limitations that have been discussed so far, and as such includes:

- Researching ways of adding support for type inference of the JavaScript standard constructs *Function* and *Array*.
- Building upon the Flycatcher system by developing a more 'clever' candidate test generator. The improved test generator could use search-based heuristics as opposed to a random strategy to explore the search space of possible tests, as is done in RuTeG [26].

An other interesting extension would be to generate tests that allow for parameters to be of a type that is user-defined but inaccessible. This could be implemented using the *Proxy* object discussed at length in this report, to substitute the missing classes by trapping all of the expected methods (but only those) and responding with the expected behaviour. However, note that using this strategy assumes that the tester would run their tests in an environment that supports Harmony *Proxies*.

Lastly, so that early adopters can try out the tool's current functionality and in order for this future work to materialise, we aim to release Flycatcher as open-source within the large and enthusiastic Node.js online community in the near future.

Appendix A

Code

A.1 Linked list

```
// JavaScript linked list
// Copyright (c) 2007 James Coglan

// It's MIT-licensed, do whatever you want with it.
// http://www.opensource.org/licenses/mit-license.php

var Node = function(data) {
  this.prev = null; this.next = null;
  this.data = data;
};

function LinkedList() {
}
LinkedList.prototype = {
  length: 0,
  first: null,
  last: null,

  append: function(node) {
    if (this.first === null) {
      node.prev = node;
      node.next = node;
      this.first = node;
      this.last = node;
    } else {
      node.prev = this.last;
      node.next = this.first;
      this.first.prev = node;
      this.last.next = node;
      this.last = node;
    }
    this.length++;
    return true;
  },
```

```
remove: function(node) {
  if (!node.prev) return false;
  if (this.length > 1) {
    node.prev.next = node.next;
    node.next.prev = node.prev;
    if (node == this.first) { this.first = node.next; }
    if (node == this.last) { this.last = node.prev; }
  } else {
    this.first = null;
    this.last = null;
  }
  node.prev = null;
  node.next = null;
  this.length--;
  return true;
},

size: function() {
    if(this.first) {
        this.first.data;
    }
    return this.length;
},

prepend: function(node) {
    if (this.first === null) {
      this.append(node);
      return;
    } else {
      node.prev = this.last;
      node.next = this.first;
      this.first.prev = node;
      this.last.next = node;
      this.first = node;
    }
    this.length++;
  },

  insertAfter: function(node, newNode) {
    newNode.prev = node;
    newNode.next = node.next;
    node.next.prev = newNode;
    node.next = newNode;
    if (newNode.prev == this.last) { this.last = newNode; }
    this.length++;
  },
```

```
    insertBefore: function(node, newNode) {
      newNode.prev = node.prev;
      newNode.next = node;
      node.prev.next = newNode;
      node.prev = newNode;
      if (newNode.next == this.first) { this.first = newNode; }
      this.length++;
    },

    at: function(i) {
        i+1;
        if (!(i >= 0 && i < this.length)) { return null; }
        var node = this.first;
        while (i--) { node = node.next; }
        return node;
      },

      randomNode: function() {
        var n = Math.floor(Math.random() * this.length);
        return this.at(n);
      }
}
```

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] ALSHRAIDEH, M. A complete automation of unit testing for javascript programs. *Journal of Computer Science 4*, 12 (2008), 1012–1019.

[3] 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.

[4] 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.

[5] ARTZI, S., DOLBY, J., JENSEN, S., MØLLER, A., AND TIP, F. A framework for automated testing of javascript web applications. In *Proceeding of the 33rd international conference on Software engineering* (2011), ACM, pp. 571–580.

[6] 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.

[7] 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.

[8] CUSTEM, T. V. harmony:proxies. `http://wiki.ecmascript.org/doku.php?id=harmony:proxies`, 2011.

[9] DUCASSE, S., ORIOL, M., BERGEL, A., ET AL. Challenges to support automated random testing for dynamically typed languages.

[10] ECMA. 262: Ecmascript language specification, 2011.

[11] 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.

[12] FLANAGAN, D. *JavaScript: the definitive guide*. O'Reilly Media, 2006.

[13] FULGHAM, B. The computer language benchmarks game. `http://shootout.alioth.debian.org/u32/which-programming-languages-are-fastest.php`, 2012.

[14] 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.

[15] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: directed automated random testing. In *ACM Sigplan Notices* (2005), vol. 40, ACM, pp. 213–223.

[16] 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.

[17] 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.

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

[19] HEIDEGGER, P., AND THIEMANN, P. *Contract-Driven Testing of JavaScript Code*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 154–172.

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

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

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

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

[24] LLC, D. Programming language popularity. `http://www.langpop.com`, 2011.

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

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

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

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

[29] 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.

[30] MICHAEL, C., MCGRAW, G., AND SCHATZ, M. Generating software test data by evolution. *Software Engineering, IEEE Transactions on 27*, 12 (2001), 1085–1110.

[31] MYERS, G., SANDLER, C., AND BADGETT, T. *The art of software testing*. Wiley, 2011.

[32] 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.

[33] 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.

[34] 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.

[35] PRASANNA, M., SIVANANDAM, S., VENKATESAN, R., AND SUNDARRAJAN, R. A survey on automatic test case generation. *Academic Open Internet Journal 15* (2005), 1–5.

[36] RAMAMOORTHY, C., HO, S., AND CHEN, W. On the automated generation of program test data. *Software Engineering, IEEE Transactions on*, 4 (1976), 293–300.

[37] 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.

[38] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 513–528.

[39] SEN, K., MARINOV, D., AND AGHA, G. *CUTE: A concolic unit testing engine for C*, vol. 30. ACM, 2005.

[40] TAHBILDAR, H., AND KALITA, B. Automated software test data generation: Direction of research. *International Journal of Computer Science and Engineering 2*.

[41] TILLMANN, N., AND DE HALLEUX, J. Pex–white box test generation for. net. *Tests and Proofs* (2008), 134–153.

[42] TIOBE. The top 10 programming languages, 2011.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.