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

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

List of Figures

List of Examples

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 [29]. 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 [23]. Yet, until very recently, dynamic programming languages had been left out of the equation — but their increasing popularity and a renewed interest in them prompts the need to start including them in the automatic testing research effort.

One such programming language that has been growing in popularity in the past few years is JavaScript, with new frameworks and libraries 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 [16], 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.

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. 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 made difficult by the absence of static types.

Contribution

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. 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.

[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

Chapter 2

Background

In this chapter we will start by giving an overview of software testing, with particular emphasis on aspects of it that are relevant to this project. We will then take a look at the state of the art in automatic test data generation (ATDG) in order to understand the approach that will be used for Flycatcher. Because the tests will be object-oriented, method call sequences also need to be generated for the test cases and we will look at the state of the art for doing that too. Finally, we will further justify our choice of JavaScript and describe features of it that are of importance for this project.

2.1 Dynamic software testing

2.1.1 Overview

We can define the activity of dynamic testing as testing that requires execution of the software with test data as input [17] and characterise it with respect to three parameters namely the amount of knowledge assumed by the tester, the target of the tests and the stage of development at which they are executed. The amount of knowledge of the software under test can be divided into three categories: structural (white-box testing) testing, functional (black-box testing) and a hybrid of the two (grey-box testing). The target of the tests refers to their granularity, from testing specific units of code (unit testing) to an entire integrated system (system testing). The stage at which the tests are undertaken determines whether they are regression tests, alpha-tests, beta-tests, acceptance tests *etc*. With Flycatcher, we 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 [19] 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 [7], 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 [7].

2.1.3 Unit testing

Unit testing consists in testing individual and independently testable units of source code [21]. 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 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 [21]. 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 [17], many classifications exist for ATDG techniques. For our purposes, the first distinction that we need to make is between white-box, black-box [25] 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 [17, 7, 29]:

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

The other classification of white-box ATDG concerns the type of implementation: *static*, *dynamic* or a *hybrid* of the two [11, 19]. We will focus on the latter 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 [29]. Figure 2.1 summarises what we believe is an intuitive characterisation of ATDG techniques with respect to this project and the one which will guide our choice of implementation. Many techniques can be found under each

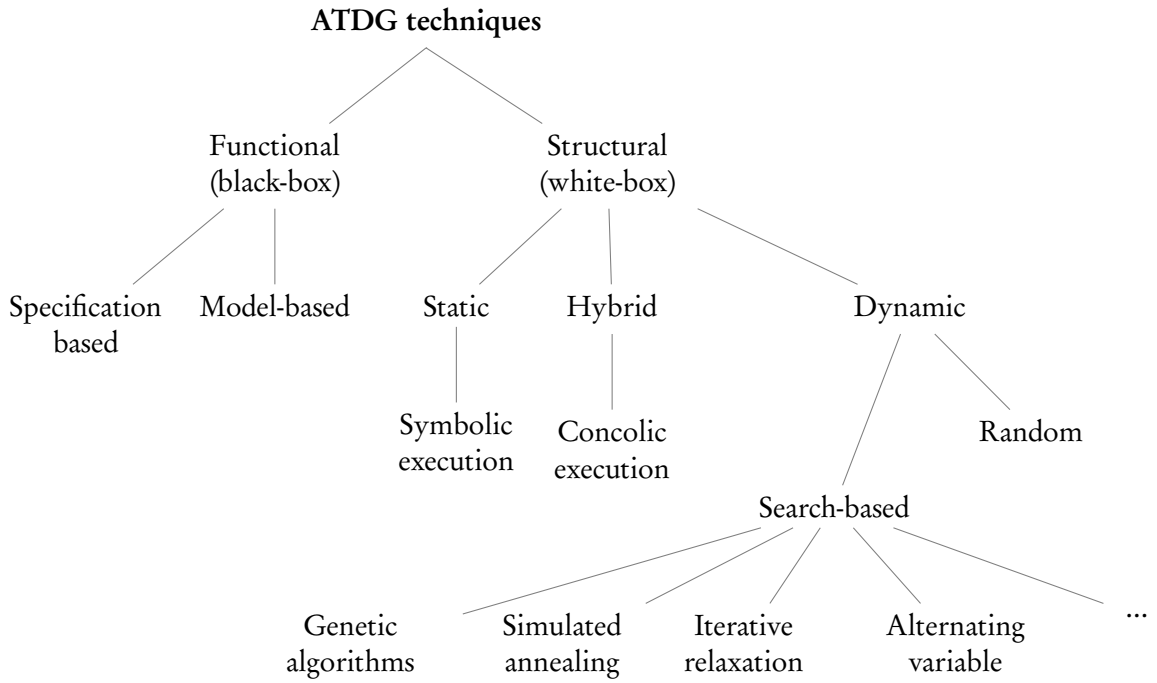

Figure 2.1: Overview of ATDG techniques

of the static, dynamic and hybrid implementation categories and we will only list the most noteworthy to us.

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

2.2.2 Static test data generation

Static structural test data generation is based on information available from the static analysis of a program, without requiring that the program be actually executed [19]. Static program analysis produces control flow information that can be used to select specific execution paths in order to try and achieve good coverage. The goal of ATDG is then to generate data that executes these paths.

Every time control flow branches, *e.g.* at `if` statements, there is a corresponding predicate or branch condition. These predicates can be collected along a path and conjoined to form the path predicate. By solving the path predicate in terms of the input variables, we can obtain test data that executes that path. However, in order to rewrite the path predicate in terms of the input variables we need to take into account the execution of the program. Hence, to generate

the test data statically a technique called symbolic execution [13] is used.

Symbolic execution gathers constraints along a simulated execution of a program path, where symbolic variables are used instead of actual values, such that the final path predicate can be rewritten in terms of the input variables. Solving the resulting system of constraints then yields the data necessary for the traversal of the given path [12, 13]. There are a lot of technical difficulties associated with symbolic execution [7, 20, 19]:

- 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 [26, 10, 22], 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 [23] — we will cover this type of ATDG implementation in 2.2.3. Due to the numerous problems posed by purely static ATDG, its weakness with dynamic types and constructs [7, 29] and the complexity of building a fully-fledged symbolic executor for a language [7, 11], 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* [23]. 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 inputs are used when need be. This

[1] the execution paths followed during the symbolic execution of a procedure [13]

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 [9], CUTE [28], JPF-SE [2], PEX [30], EXE [5] and KLEE [4].

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. 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 with, possibly randomly, selected input and feedback is collected at runtime regarding the chosen coverage objective [7]. The feedback is usually obtained through some form of instrumentation of the program that monitors the program flow. Inputs can be continually generated randomly, relying on probability to achieve the coverage objective — this is known as *random* test data generation and does not perform well in general [7]. On the other hand, inputs can be incrementally tuned based on the feedback (using different kinds of search methods) in order to satisfy the coverage objective — this is known as *search-based* test data generation [19], where the search-space is the control flow graph of a program. The main drawback of dynamic ATDG is that it is reliant on the speed of execution of a program and as the number of required executions to achieve satisfactory coverage may be high, this leads to an overall expensive process. Below we will present the random and search-based approaches in more detail, in order to understand which would be more suitable for Flycatcher.

Random approach

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

2.2.5 Challenges of dynamic languages

Most of the research on ATDG so far concerns static programming languages [17] 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 [6].

Generating test data of the required type

Given that method parameters do not have static types in dynamically typed languages, we do not know what arguments to pass to them. A potential solution to this is to use a method called *type inference* [24], 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 [18], 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 [14, 33, 27, 34], but all these approaches focus on statically typed languages (C/C++), require static program analysis and mostly lack generality.

Another approach uses needed-narrowing [3] or lazy instantiation [15] — where instances are created empty 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 type test data generation in Flycatcher, we will use a different method but the idea of returning adequate objects upon their use in the program 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 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 generator, so that preconditions for the program are respected.

As for real software errors, Flycatcher will deal with those too, and this matter will be revisited later.

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

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

Because it is impossible to know how the application will use the class/object under test in practice, as many relevant test cases as possible must be tried in order to maximise the likelihood of finding a bug inside the class in question. 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 can be used, as for the generation of input *data*, as a search objective to guide the generation of *method call sequences* for our test cases. Tonella was one of the first

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

to use search-based methods for the generation of adequate object-oriented test cases, using genetic algorithms as the search heuristics method [32]. Indeed, the procedure of using genetic algorithms to generate input test data for Flycatcher described in **??** can simply be extended to the generation of object-oriented tests by adapting individuals so that they represent the structure of a test case.

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

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 software techniques: unit testing, regression testing *etc.* Due to the relative recency of the surge in popularity of JavaScript, which can be observed in Figure 2.2, no effort has yet 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.

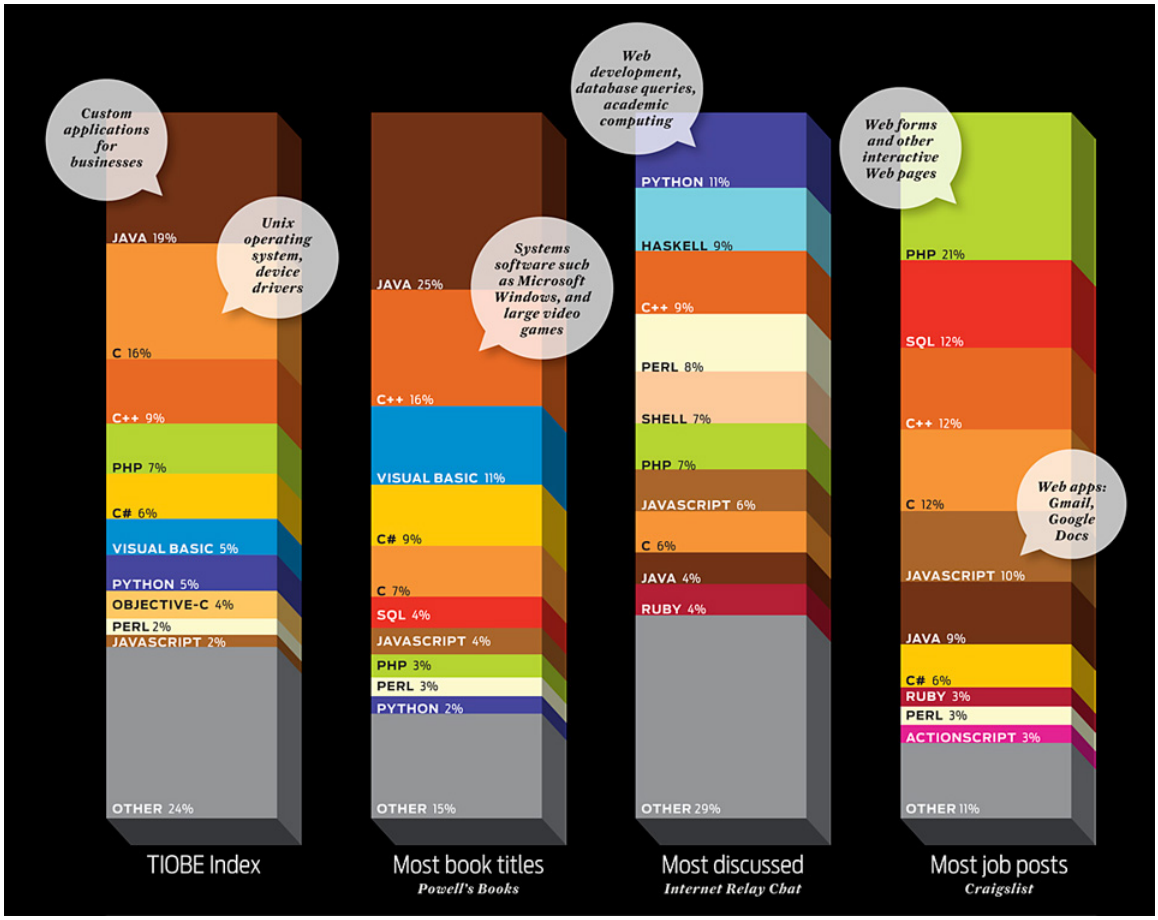

Figure 2.2: The top 10 programming languages 2011 [31]

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 through the use of 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[3] — although given the time constraints, we will work with a subset of it.

2.4.3 Idiosyncratic features

For the readers who are not familiar with the language, we present the characteristic features of JavaScript [8] 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:

[3] http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

```
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, but rather that 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[4] 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)

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 ignore them at this stage.

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

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, but it is not necessary to detail it here. It is worth noting about Proxies, as this will matter to us later, 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.

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. To illustrate 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 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 include it in the research work on automatic test generation. 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. Random test data generation can be improved by search heuristics and genetic algorithms will be our heuristic method of choice. 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

Design

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 components and give an overview of the system.

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. Thankfully this need was partially met by the development of *Node.js* that started in 2009. 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 Components

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.

3.2.1 Analyser

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.

3.2.2 Test Generator

The next component involved is the **Test Generator**, which has two implementations consisting of a random test generator to start off with followed by a search-based generator for improved efficiency. Naturally, the more runtime information we have on the class under test (CUT) the more accurate the tests can be. Hence, the initial tests are not accurate but serve to gather runtime information, until we have sufficient information to generate tests with confidence that they are appropriate. As such, this component is tightly coupled with an-

other, which orchestrates a virtual runtime environment in order to collect the necessary runtime information: the **Executor**.

3.2.3 Executor

The **Executor** component has two major responsibilities:

1. providing a bespoke runtime environment that enables the collection of information concerning the type of the parameters of a method under test (MUT)

2. tracking the code coverage achieved by the generated tests, in order to measure their quality

The **Executor** and the **Test Generator** thus work in to-and-fro until either satisfactory code coverage is achieved or a termination criterion is met. Upon termination, the tests that were deemed accurately typed and that contributed to code coverage are collated into a suite of unit tests and 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*.

3.3 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.

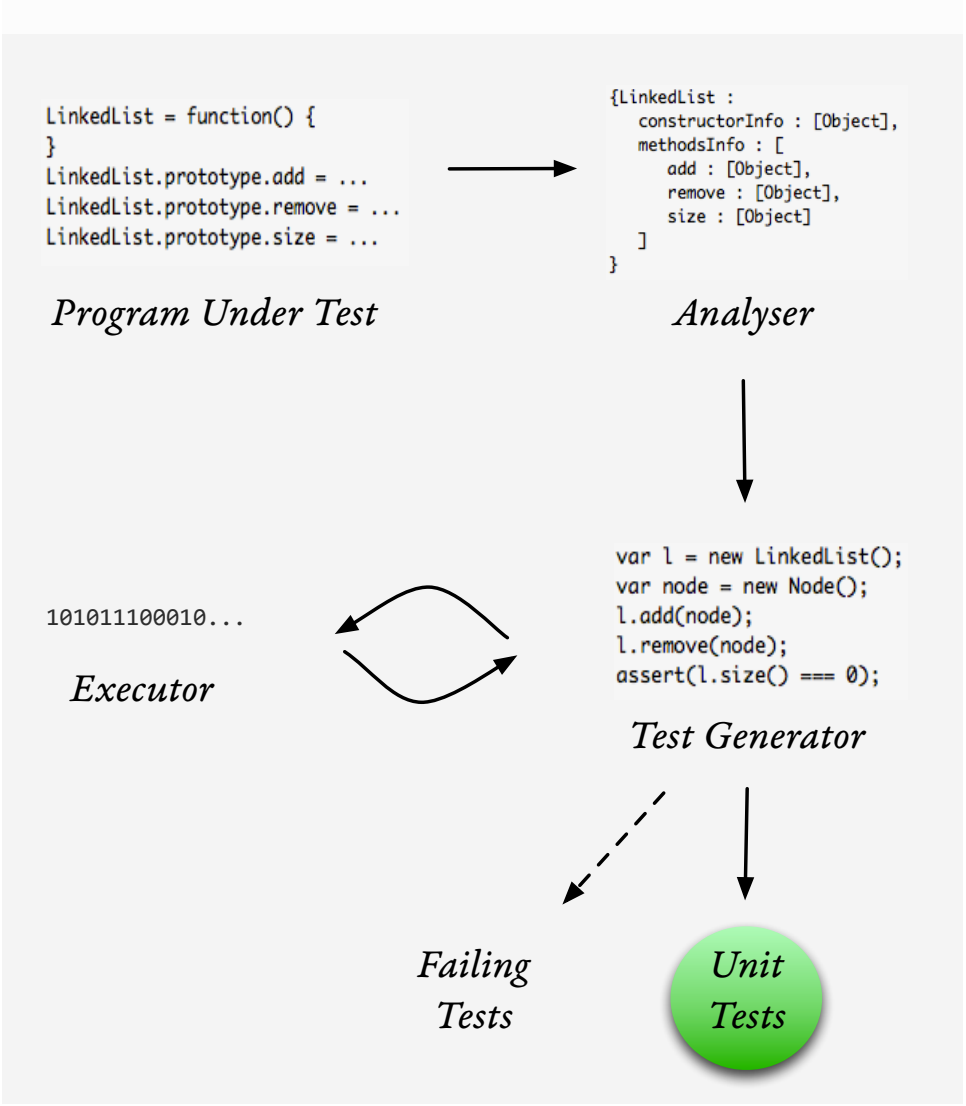

Figure 3.1: The Flycatcher system

Chapter 4

Analyser

The targets of our unit tests are classes, understood as the definition 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 the 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 be nonsensical — it 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 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 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 has the option of *specifying* the namespaces of interest, but the underlying mechanism for extracting information is exactly the same.

Additionally, the CUT and its dependencies may be spread over multiple files. In response to this, Flycatcher users have the option of specifying multiple input source files, but this also has no implications.

What is necessary however is for any classes that *are* used by the CUT to be accessible in the provided code and namespaces, 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 we have loaded the source code under test into the sandbox, we can iterate through the appropriate namespaces and retrieve 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 class constructors is straightforward but to retrieve that 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 the constructor. If we do pass wrong parameters though, such as passing a number when an object is expected, the constructor will crash and we won't be any closer to retrieving information about the 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 it Proxy objects 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 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

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

sound choice as it can't make any of the primitive operators effect a crash (but for instance `"string"++` would break). The Proxy API offers many more traps than the `get` trap, but we focus on the `get` trap for clarity's sake, as the behaviour of the other traps is analogous to what we have described. 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,
        function() {
            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 doesn't crash. The purpose of this process is to find out the signatures of the class's methods and these cannot be affected by the class parameters — except for the case where *the class method itself* is passed in as a parameter, which we do not deal with.

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:

```
// 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).

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:

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

In fact however, as the `ProgramInfo` object is the central element of Flycatcher, it is more complex than that. But its additional features will be revealed in the later chapters, where relevant.

Chapter 5

Random Test Generator

In this chapter we detail the process for generating *random* tests, in the hope of achieving 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 for the parameters are correct. Finally we discuss the formatting of those tests, with regard to whether they are intended for the **Executor** or as final output.

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. Instantiate the object
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

Unit test format

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: Unit test format

Executor format

For all the results of calling the MUT in a candidate test to be recorded, so that the assertions seen in example 5.1 can be created, the format of test 5.1 in the **Executor** is in fact:

```
(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 declarations

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. So, in the example,

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

The number of method calls made in an attempt to modify the state of the CUT instance is chosen *at random*, with a configurable maximum. 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 to exercise full coverage, as its behaviour depends on whether the 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 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 work with antecedent parameters and not necessarily redefine its own, it is impossible to hand it such a reference. 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 already exists in the test. Upon a series of experiments, which will be detailed in the evaluation section, it was determined that a 25

Although we have discussed how tests are constructed, we have purposely ignored one important detail: *we do not know the types of any function parameters*. How do we know that the `LinkedList.add` method takes an object of type `Node`?

5.2 Types

In the example of the previous section, 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, we create a new special type of object: 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 will eventually be substituted with objects of the appropriate type. Initially however, this is what a test might 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). Because they are not meant as output unit tests, they also do not contain assertions. In

fact, the format should resemble example 5.2, but we simplify it here for clarity.

To carry on with the example, 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 5.1.

5.2.2 Type inference

The information that is collected when tests are run by the **Executor** is stored in the `ProgramInfo` object introduced in chapter 2. Upon each **Test Generator/Executor** iteration for that test, we update the `ProgramInfo` object's type information using the new 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 a property on that parameter. 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 class has fields or methods that match the member accesses on the parameter, the one with the highest correspondence is elected. But if

there is a member access with no matches[1], this means that we are faced with a class which is not accessible in the scope of the program and we abort the test generation process for that particular method with a warning.

When we do the `ProgramInfo` type inference updates, the question of whether a parameter has any member accesses comes *first*, before we look at what sort of primitive it is the most likely to be. The reason behind this is simple: any member access rules out the possibility of a parameter being a primitive, since by definition primitives in JavaScript have no properties.

If and only if no member accesses were recorded, do we look at the parameters *primitive score*.

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 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* and *undefined* and consider that *boolean* values can be substituted by `0` or `1`. The primitive scoring object thus corresponds to:

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

Example 5.7: Primitive scoring

Delaying updates

Because we do not want to jump to conclusions to quickly when inferring types using the member accesses and primitive score, we do not start updating the `ProgramInfo` object until either:

1. We have collected sufficient information to make a confident decision

   or

2. A sufficient number of calls *involving* that parameter have been made

Once more, the optimal values for those variables are highly dependent on the program under test, hence they are configurable. We nevertheless provide a sensible default based upon experimentation of, respectively:

1. members called > 2 *or* max(primitive score) > 10

[1] except for the hidden properties inherited from `Object` in JavaScript

2. number of calls involving that parameter > 15

Note that the two conditions are a disjunction, they do not need to both be true. Also, even if the first one is never realised, the second will necessarily be realised after a certain number of **Test Generator/Executor** iterations for that test.

Finally, there is one last case to deal with: when the second condition is met but we *still* have no information to work with *i.e.* our parameter's properties were not accessed nor was it used in an operation. In such a scenario we assume that given that the test was run a reasonable amount of times (again, this is configurable if the default is not appropriate), it is likely that the parameter is never used. Hence, we replace it with a random primitive, in order to remove any `Unknowns` from the candidate test, so that it can be output.

The reasoning is that if a parameter is not used in any way, whatever value we give it has no impact whatsoever. We make the design choice of substituting the unused parameter with a random value rather than aborting the test generation for that method. In the *unlikely* event that the unused parameter is in fact used but just happened not to be in the minimum number of runs chosen, tests will fail and the user can see why in the logs. Upon seeing that the parameter was inferred out of lack of information (a warning is issued), they may then want to adjust the configurable termination parameters to suit the needs of their program.

5.3 Outcome

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: contains no `Unknowns`

A test may contain no `Unknowns` but 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 forever as it may not achieve the expected coverage setting for a MUT. Hence, there are two *liveness* safeguards for this scenario:

1. An optional timeout after which Flycatcher stops generating tests for a MUT
2. Abort after a specific number of test runs during which coverage did not improve

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 liveness safeguards will also apply, however the tests that do fail will be recorded. These tests are output in a log file, such that the user can diagnose and fix failures in their program[2].

If termination is not caused by timeouts, then Flycatcher stops generating tests for a MUT when the desired coverage, initially specified by the user, is reached[3]. At that point, all of the elected tests are concatenated into a unit-testing suite and output in the format of the user's preferred unit-testing framework. The unit-testing formats available are:

- node-unit
- expresso
- vows

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.

[2] it is also possible that the errors are due to Flycatcher making type inferences too early on and getting it wrong, which the user will be able to determine from the failing test logs

[3] the default is full coverage, 100%

CHAPTER 6

Executor

The tests that are generated by the **Random 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 'best' ones

To those ends, a bespoke runtime environment was created: the **Executor**. In this chapter, we elaborate on how runtime type information is collected and how 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, that we shall name `RecordingProxy` to be precise, tailored for type information collection. The information is collected inside the `ProgramInfo` object, as discussed in chapter 5. Hence, the `RecordingProxy`'s handler needs to be instantiated with information that lets the proxy 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 stand 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 easily recorded using the `get` trap of the `RecordingProxy`: 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 `RecordingProxy`'s handler, where in `ProgramInfo` to store that information.

When a property is accessed however, the `get` must return an appropriate object for the 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 `RecordingProxy` type — the proxy we return no longer stands for a parameter and as such does not collect any type information (it's role is simply not to crash). The only difference is that we randomise 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.

6.1.2 Accumulating primitive scores

The primitive operations that involve an `RecordingProxy` are also recorded using the `get` trap, as they result in the internal `valueOf` function being called. Hence, the handler need only compare the name in `handler.get(receiver,name)`

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

to `valueOf` to determine whether it is dealing with a primitive operation. Where this becomes more involved however, 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 the 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`[2] 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 6.1 shows the hints that are looked out for and by how much the score of a primitive type is increased when a hint is encountered. We recall that the only primitive types considered are *number* and *string* as we consider that the *boolean* values `true` and `false` can be substituted by `1` and `0` respectively. 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*.

Hint	*number*	*string*
`++, --`	10	0
`>, <`	2	1
`-, *, /, %, ≪, ≫, ⋙, !, ^, |, &, ~`	2	0
`[0 - 9]`$^+$	5	0
`"..."`	0	5

Table 6.1: 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.

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 **Random Test**

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

Generator uses it to try and infer a type for that parameter. Having discussed the collection of type information, we move on to discussing the second mission 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, we have to resort to weaker coverage measures such as *branch coverage* which tracks the truth values at every branching statement, or simply *statement coverage*, which tracks the statements that are executed. In Flycatcher we chose to implement the basic measure of *statement coverage*, in order to focus the bulk of our efforts on the test generation process itself.

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 generated tests 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. For example, when using code coverage as a heuristic to *improve* the tests being generated for a MUT, coverage in some other remote part of the program is not relevant. 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, in the source code that is run by the **Executor**, *all the statements* in the definition of the MUT 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[3]. Thanks to that package, wrapping is made relatively simple:

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

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

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

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

Example 6.1: Wrapping with burrito

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

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

At the end of this wrapping process, the full array of indices used thus represents all of the MUT's statements. That array can be used to keep track of coverage.

6.2.2 Implementing the callback

The **Executor** must implement the coverage callback which is called when a statement is executed. The implementation itself is trivial: the callback must simply set the value of the node index it is called with to `true`. For example, if the MUT has nine nodes/statements, at the beginning of its test generation process, its coverage tracker is:

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

If the first and last node are executed, it becomes:

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

As a result, the **Executor** can report:

- 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 in the various termination criteria of the test generation process for that method

Chapter 7

Evaluation

CHAPTER 8

Conclusion

Appendix A

Examples

BIBLIOGRAPHY

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

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

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

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

[5] CADAR, C., GANESH, V., PAWLOWSKI, P., DILL, D., AND ENGLER, D. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC) 12*, 2 (2008), 10.

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

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

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

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

[10] GOLDBERG, A., WANG, T., AND ZIMMERMAN, D. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis* (1994), ACM, pp. 80–94.

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

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

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

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

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

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

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

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

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

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

[27] 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.

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

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

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

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

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

[33] 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.

[34] 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.