Master's Thesis

Testcase Generation based on Property-based Testing

submitted by

Jan de Mooij

on August 19, 2011

University of Amsterdam
Master Software Engineering

Centrum Wiskunde & Informatica
Dutch National Research Institute for Mathematics and Computer Science

Supervisor: Prof. Dr. D.J.N. van Eijck

Abstract

JavaScript is an increasingly popular programming language for writing web applications. While JavaScript was originally developed as a scripting language for adding dynamic behavior to websites, nowadays JavaScript is used for writing complex applications like MP3 decoders or even x86-emulators. To make this possible browser vendors are implementing advanced interpreters and JIT-compilers. The downside of this "performance race" is that new browser versions often have new regressions, despite a large number of existing test frameworks. We have written a new JavaScript engine test framework based on property-based testing. The tool has found both correctness bugs and crashes in all major browsers.

Preface

This thesis is the result of the research project I did for the master Software Engineering at the Centrum voor Wiskunde en Informatica in Amsterdam.

I would like to thank Jan van Eijck for giving me the opportunity to work on this project and for his valuable feedback and supervision. I also want to thank him for teaching the course Software Testing which introduced me to property-based testing and inspired this project.

I also want to thank Mozilla and Opera for their kind replies to my bug reports and their interest in this project. I want to thank Jesse Ruderman and Christian Holler from Mozilla's security team for giving me access to their testcase reduction tools.

Finally, I would like to thank my family and friends, especially my parents, for their support and encouragement.

Contents

Chapter 1

Introduction

JavaScript is becoming increasingly popular for writing web applications. Nowadays every browser comes with an optimized JavaScript engine. While JavaScript was initially only used for adding dynamic behavior to websites, modern web applications heavily depend on fast JavaScript execution. Recently there've been ports of large applications to JavaScript, including MP3 decoders, JPEG encoders and even x86 emulators.

The downside of this is that modern JavaScript engines are becoming more and more complex. Now every browser comes with an advanced JIT-compiler (some browsers even have multiple compilers) and other optimizations. Even regular expressions are compiled to efficient machine code. [9]

All these optimizations and performance improvements can introduce new bugs or differences between implementations. These problems are often found months later. As a result, some websites may no longer work correctly in a particular browser version. Even more serious are security bugs which may be exploited by hackers. Google en Mozilla pay hundreds of dollars for information about code defects that can be exploited [6, 11].

In this thesis we will describe our system for automatically generating correctness tests. We will first look in more detail at our reasons for choosing JavaScript, give an overview of the related work and formulate the research question (section 2). We then describe how we can use property-based testing to generate test cases (section 3). After this we give an overview of our implementation (section 4) and present our results (section 5). We then suggest directions for further work and summarize the contributions of this thesis (section 6).

Chapter 2

Motivation

2.1 JavaScript

The test case generation approach described in this thesis is language independent. However, we decided to focus on JavaScript for our implementation for the following reasons:

1. **Multiple implementations**. There are at least five widely used JavaScript implementations, the engines used in the web browsers Internet Explorer, Firefox, Chrome, Opera and Safari. It's important for implementations to be compatible because differences in behavior may prevent a website from working in a particular browser.

2. **Standardized**. JavaScript is based on ECMAScript. ECMAScript is standardized by Ecma International in the ECMA-262 specification [13]. This means tests can be written or generated based on the specification without the need to reverse-engineer a particular implementation. The specification also makes it easier to report bugs to browser vendors as most bugs can be traced back to a single paragraph or algorithm in the specification.

3. **Widely used**. All popular web browsers have builtin support for JavaScript. This means every modern computer and smartphone has a JavaScript engine. Popular websites like Gmail, Twitter and Facebook use JavaScript extensively.

4. **Performance race**. Nowadays all major browsers have advanced JIT-compilers for executing JavaScript. Mozilla Firefox 4, for in-

stance, has an interpreter and two JIT-compilers, a method-JIT and a tracing-JIT [7]. Performance optimizations can introduce complexity and bugs. An example of this is the number type. ECMAScript specifies that numbers are doubles. However, for performance reasons modern JavaScript engines internally also represent numbers using 32-bit integers. This introduces complexity because the engine now also has to support (arithmetic) operations on integers and has to handle integers overflowing to double et cetera [8].

5. **Complex language**. JavaScript has a number of features that are hard to implement correctly and efficiently. The language is dynamically typed, builtin-functions can be replaced and object properties can be added or deleted at any time. This makes it hard to implement JavaScript efficiently. A recent example of this is an optimization Microsoft added to (a development version of) Internet Explorer 9: dead-code elimination could remove certain (arithmetic) operations but this optimization was unsound if one of the operands is an object [3].

6. **Many existing tests**. There are many existing test suites for testing JavaScript engines. Recently the ECMAScript committee started developing its own test suite, test262. The current version (0.7.5.2) already has 10,942 tests (based on the ECMAScript specification) from various sources. [1]

7. **Other testing techniques**. While fuzzing is most often used to find crashes or security bugs, Mozilla also uses fuzzing to find correctness bugs by running randomly generated scripts first in the interpreter and then in the JIT-compiler. This makes it possible to find bugs in the JIT-compiler by comparing the output of the two runs.

To summarize, JavaScript is an interesting language to generate tests for because there are a number of different implementations (most of them heavily optimized) and compatibility is important. On the other hand many websites use JavaScript and there are many existing tests. This means regressions are often found soon because either a test fails or a website stopped working. This means JavaScript is probably one of the most challenging languages for automatically finding correctness bugs.

2.2 Test generation methods

There are a number of methods currently in use for generating test cases. In this section we will discuss some of the more widely used techniques and

how they can be used for testing JavaScript engines.

2.2.1 Fuzzing

Fuzzing is a popular technique for finding security bugs in software. A JavaScript fuzzer generates random source code and then runs the file in the interpreter. Scripts should never be able to crash the interpreter, so if the interpreter *does* crash there must be a bug somewhere. Fuzzing is popular because writing a fuzzer and generating tests is easy and cheap.

Introducing jsfunfuzz
In 2007 Jesse Ruderman of Mozilla wrote jsfunfuzz, a fuzzer to generate random JavaScript files. jsfunfuzz has found many bugs in Mozilla's JavaScript engine and is also used extensively by other browser vendors to catch regressions. Mozilla also uses this fuzzer for correctness testing by running the script with different compiler settings and comparing the output. This technique can be useful for finding bugs in the JIT-compilers but is unable to detect bugs in runtime functions because these use the same code independent of the JIT-compiler options. Comparing the output against a different JavaScript is hard since the test may depend on implementation-specific features like enumeration order.

Announcing cross_fuzz
In this post Zalewski presents cross_fuzz, a fuzzer for finding bugs in web browser DOM bindings. This fuzzer does not create JavaScript code but randomly creates DOM objects and passes them to other functions. This fuzzer has found several security bugs in all major browsers. One problem with this approach is that it can be hard to reproduce crashes because the fuzzer does not generate source code which can be run multiple times.

Grammar-based whitebox fuzzing [10]
This paper describes several techniques for fuzz-testing JavaScript implementations. The authors then introduce *grammar-based whitebox fuzzing*. This technique uses symbolic execution and constant solving so that it's able to find more problems. They also measured code coverage of the code generation engine found in Internet Explorer 7 to measure its efficiency. Coverage increased from 61% to 81%. Unfortunately this paper does not discuss whether they used their framework for correctness testing.

2.2.2 Property-based testing

Property-based testing is a technique for finding correctness bugs in software. Instead of writing explicit test cases, more generic *properties* are formulated. The testing framework then generates a large amount of data and tests whether the property holds.

QuickCheck: A lightweight tool for random testing of Haskell programs [5]
This paper describes QuickCheck, a property-based testing framework for Haskell. Properties are written as Haskell functions and the framework will call these functions with randomly generated data. It's possible to generate random data for user-defined types by writing *generators*. Although QuickCheck was written for Haskell, QuickCheck has also been ported to other languages such as JavaScript and Java.

Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values [16]
SmallCheck and Lazy SmallCheck are testing frameworks for Haskell comparable to QuickCheck. The main difference is that SmallCheck does not generate *random* data but starts with small values and then increases the value. An advantage of this approach is that it's deterministic, the test input is always the same. According to the author the tools are complementary and SmallCheck is not meant to be an alternative for QuickCheck.

2.3 Research question

In the previous sections we have described various testing techniques. The advantage of fuzzing is that it's easy and cheap to generate random programs, but it's hard to find correctness bugs. Property-based testing frameworks allow us to write down properties, generate random input for them and check correctness. Instead of generating random data, however, we want to generate random *source code*. This has a number of benefits, for instance it makes it possible to test more parts of the JavaScript engine. Our reasons for this approach will be further described in section 3.1.

Based on this we can formulate the following research question: *"Can property-based testing be used to find new bugs in JavaScript engines?"*

Because tests generated using property-based testing are more generic than the existing correctness tests, our hypothesis is as follows:

1. The tests are more generic, so many of the existing tests will be a subset of the tests generated by our framework.
2. Our test framework will find new bugs because we can generate a large amount of testcases.
3. Generating random source code instead of random data allows us to find more bugs.

Chapter 3

Background

3.1 Property-based testing

The idea behind QuickCheck is straight-forward: pass random data to a test function and ensure the property always holds. There have been ports of QuickCheck to JavaScript [17]. These frameworks work more or less as follows: the user defines a function, annotates it with the expected types and the framework will then call this function with random data. This approach works very well for a functional language like Haskell and could be very useful for testing builtin JavaScript functions because the return value depends mostly on the argument values. It is less useful for testing other parts of the engine because even though the tests use random data, the program always contains the same statements and expressions.

So ideally the framework should go a step further. It should be able to generate real *testcases*: random JavaScript files with embedded correctness checks. This way, we can check that properties not only hold in a particular case, but also when placed inside a loop, when used in the global scope, inside eval, et cetera.

We can accomplish this by generating random *source code* instead of random *values*. This has a number of advantages over using a JavaScript port of QuickCheck:

1. We can separate the "test generation" and "test running" steps. This means we can generate JavaScript test files and run them in other browsers or shells. This also allows us to use non-standard JavaScript extensions in the test generator.

2. Properties can be tested inside loops and other contexts. This allows testing loop optimizations, trace compilers et cetera. Properties can also be tested in the global scope, this is important for JavaScript because global variables can be accessed as properties of the global object and therefore have to be treated differently.
3. Multiple different properties can be combined in the same script or function.
4. We can test more parts of the JavaScript engine. For instance, generating source code allows us to better test the parser, bytecode emitter and register allocator.
5. We can add strict-mode declarations to functions or scripts to test *strict mode*, a new feature in ECMAScript 5.

3.2 Test properties

The aim is to run tests in both the shell and the browser because not every browser vendor does release separate JavaScript shell builds. However, browsers use time-outs to prevent long-running scripts from "freezing" the browser. Even though this time-out can be disabled in some browsers, long-running tests are still undesirable: we would need to kill the browser when a script runs for more than a few seconds, making the test runner slower and more complicated.

Furthermore, we want the generated tests to be *deterministic*. This means that control flow should not depend on random numbers, the current date or time or other sources of non-determinism. Non-determinism is undesirable because it makes it harder to reproduce failing tests. Reproducibility is important to determine whether a test failure is a real bug in the JavaScript engine or a bug in the test framework. This property also makes it easier to compare different JavaScript engines because every engine runs exactly the same code.

To summarize, we want to generate test cases with the following properties:

1. A test case has no infinite or long-running loops.
2. A test case has no infinite recursion.
3. A test case is deterministic.

Note that these properties are optional, they can be disabled for testing shell builds where time-outs can be disabled.

Chapter 4

Implementation

The system consists of two main parts, the test case generator and the test runner. The test runner is written in Python and invokes the test generator to generate a single test case.

4.1 Test generator

The test generator is the most important part and is responsible for creating random JavaScript files. The test generator is written in JavaScript and requires Mozilla SpiderMonkey's command-line JavaScript shell. The test generator is not portable because it depends on some SpiderMonkey extensions, most notably Reflect.parse for parsing JavaScript code [12]. The generated tests are standard JavaScript code though and can be run in other JavaScript engines, both browser and shell implementations.

The test generator generates random JavaScript statements and expressions. The generator keeps track of scopes, variables, types and properties. This is needed to ensure that all operations are valid and tests don't throw unexpected exceptions.

4.1.1 Loops

Some JIT-compilers, like Mozilla's trace compiler, only work on loops. This means that it's important to generate various loop constructs if we want maximum coverage. To prevent test cases from running for a long time (or timing out in the browser, see section 3.2) we need an upper bound on the

number of iterations. To accomplish this the generator can generate two types of loops, "simple loops" and "guarded loops". A *simple loop* is a loop like this:

```
for (var x=y; x<z; x++) {
   // statements
}
```

Here y is a random number and z is $y+random(0, maxiter)$. Inside the loop x is available as read-only variable. Since the loop can have break or return statements we don't know the exact number of iterations but we know that it will never exceed $maxiter$.

Guarded loops support random loop expressions but need a dynamic check to make sure the loop terminates:

```
var iter1 = 0;
while (true) {
  if (iter1++ > 15)
    break;
  // statements
}
```

Every guarded loop has a variable (iter1 in this case) to count the number of iterations. Since iter1 is not modified inside the loop this ensures the loop always terminates after 15 iterations.

While these guards are sufficient to prevent infinite loops, it's still possible to trigger browser time-outs with deeply nested loops. Therefore, the number of loops in a single browser test case is limited to 4, although checks (section 4.1.5) can also add loops. When running tests in the shell this limitation can be disabled because the shell does not time-out and it's easy to terminate the shell after x seconds.

4.1.2 Function Calls

The generator can generate function calls, either direct calls or calls using call or apply. To prevent infinite recursion or browser time-outs, we don't want to generate tests like this:

```
function f(x) {
    x();
```

```
}
function g() {
    f();
}
f(g);
```

To do this our implementation assigns an (increasing) number to every generated function type. A function f can only generate a call to another function g if f has a smaller number than g. Functions are first-class citizens in JavaScript so we also have to make sure that functions are never passed to functions with greater number. This makes it impossible to generate the example given above because we will never pass g to f. In other words, we can call functions defined earlier in the program but we can never call the current function or functions defined later in the program.

4.1.3 Types

The test generator assigns types to all expressions and variables. Every type has a *containsType* method to determine whether a type contains another type. This can be used to check whether the result of an expression can be assigned to a variable. Below is a list of some of the types defined by the framework:

- **AnyType**: contains all other types
- **NullType**: contains only the null type
- **NumberType**: contains all numbers
- **FunctionType**: function type, also contains argument types, return type and index (described in section 4.1.2).
- **ConstructorType**: like FunctionType but initializes an object. Also contains information about the constructed object.
- **FunctionReturnType**: contains all functions with a certain return type, this is used to generate calls.
- **ArrayType**: an array, also contains the element type
- **GenericObjectType**: an object with a number of properties, both simple properties and getters or setters.
- **BuiltinObjectType**: a builtin object like Number or Date.prototype
- **ObjectOrPrimitveType**: contains both a primitive type (like number) and its object type (like Number)
- **SetType**: contains a number of other types so that a variable, for instance, can be assigned both numbers and booleans

This list shows that our definition of type is rather broad: types are not only used for generated expressions but are also used to query variables or expression generators of a certain type. To generate function calls, for instance, we use FunctionReturnType to generate a random expression which results in a function with a certain return type. We then look at the returned function type and generate random expressions for the function arguments.

Every type also has a *generateLiteral* method to generate a literal expression of this type. For NumberType, for instance, this method returns a random number literal. For AnyType we generate a random type and return the result of calling generateLiteral on this type.

4.1.4 Expression generators

Support for some expression types is hard-coded in the generator, in particular function calls, literals, eval, variable lookup, arguments object, the ternary operator and some binary operators. The reason for this is that these expressions have complex types, need special limitations (section 4.1.2) or are hard to write using the custom expression generator mechanism described below.

Tests can also add custom expression generators by writing small JavaScript functions. Generators are written as plain JavaScript functions to make it easy to parse or reuse them in other frameworks. Here's an example generator for Math.abs:

```
function gen_mathAbs(x) {
    return Math.abs(x);
}
gen_mathAbs.args = {x: values.anyValue};
gen_mathAbs.result = values.anyNumber;
```

Generators always have a single statement, the return statement. The expression generators are parsed using SpiderMonkey's Reflect.parse extension, the return argument is extracted so that it can be used to construct random expressions. For instance, the example Math.abs generator can be used to generate the following random statement:

```
var x = Math.abs(y * Math.abs("4"));
```

Custom expression generators provide a simple but powerful way to extend

the test generator without requiring knowledge about the inner working of the test generation process.

4.1.5 Checks

Checks are used to add correctness tests to the test case. Checks are small snippets of code that are converted to statements. A typical check looks like this:

```
function check_MathAbs(x) {
    var res = Math.abs(x);
    assert(isNaN(res) || res >= 0);
    assert(equals(res, x) || equals(res, -x));
}
check_MathAbs.args = {x: values.anyValue};
```

Here *assert* and *equals* are helper functions added to every testcase as described in section 4.1.7. This check can be turned into a testcase like this:

```
function f1() {
    var a1 = 3 * g();
    var a2 = Math.abs(a1);
    assert(isNaN(a2) || a2 >= 0, "check_MathAbs");
    assert(equals(a2, a1) || equals(a2, -a1), "check_MathAbs");
}
f1();
```

The test generator has replaced all argument uses (x in this case) by a random expression of the same type. The test generator also special-cases calls to the *assert* function: it adds the name of the check as extra argument to every assert call to make it easier to generate meaningful error messages.

The test generator cannot simply substitute every reference to an argument with a random expression of the same type. It has to preserve the "semantics" of the check. Consider this equality check, for instance:

```
function check_eq(x) {
    assert(isNaN(x) || x === x);
}
check_eq.args = {x: values.anyValue};
```

This check encodes the property that every value is equal to itself, except NaN. It would be invalid to derive a test case like this:

```
assert(isNaN([1, 2]) || [1, 2] === [1, 2]);
```

This test case will always fail since every array literal creates a new array object, and the === operator will just compare the references instead of the actual object values. Another problem is that some expressions, like function calls, are not pure: evaluating them multiple times may give different values. So this test case is also invalid since f() returns a different number for each call:

```
function f() {
    return x++;
}
assert(isNaN(f()) || f() === f());
```

To solve this problem every check is analyzed when building the AST (Abstract Syntax Tree) of the source code. For every argument we record how many times it's used, whether it's used as left-hand side of an assignment, et cetera. After this we generate a random expression for each argument. Based on the recorded information we decide whether we can replace every identifier with the generated expression or else we will assign the expression to a variable and use that variable inside the check.

For eample, for check_eq given above, we record that argument x is used three times. Next we generate a random expression for x and, based on the recorded information and expression AST type, we decide whether it's safe to use this expression directly. With this algorithm we can generate valid testcases like this:

```
function a1() {
  return 3;
}
// safe to insert number literals directly
assert(isNaN(10) || 10 === 10);
// function call is not pure, assign result to a variable
var a2 = f();
assert(isNaN(a2) || a2 === a2);
```

4.1.6 Test configuration

The test generator loads a special JavaScript file which defines the test configuration. A test configuration file contains a list of disabled checks and expression generators. This mechanism is used to disable checks which trigger known bugs or to disable features which are not supported by certain JavaScript engines.

4.1.7 Output

The output of the test generator is an AST (Abstract Syntax Tree). This AST is converted to source code and printed to stdout. The generator also prepends several utility functions. These functions are available to checks and are typical test framework functions like *assert* (throws an exception if its argument is false), *equals* (recursively compare object properties) and $isNegZero$ (return true iff its argument is negative zero).

4.2 Test runner

The result of the test generator is code for a single JavaScript file, printed to the console. The console test runner invokes the test generator, saves the output to a temporary file and then runs the file in the JavaScript shell. Both the SpiderMonkey and V8 command-line shell have special flags to control optimizations, enable JIT-compilers or tune GC behavior. To increase code coverage the test runner chooses a random combination of flags for every test case. If the shell throws an error, time-outs or crashes, the output is written to the console and the input file is copied to a special directory.

The console test runner can optionally copy every generated file to a temporary directory. This can then be used for generating browser test cases. We use PHP to create a list of all JavaScript files in the test directory, send this list to the browser and the browser runs some JavaScript to load every test file in an iframe, show a progress bar and a list of test failures, et cetera.

4.3 Tests

We have written a large number of generators and checks based on the ES5 specification. Most operators and functions have their own test file with a

number of generators and checks. Our ES5 test suite consists of 123 files with 124 generators, 317 checks and 801 asserts.

We have also written tests for some features not (yet) part of ES5 like typed arrays (used for canvas and WebGL) and WeakMap (defined by the next version of the ECMAScript specification). The test configuration file described in section 4.1.6 is used to disable these tests if the implementation does not support them.

Chapter 5

Results

We have found bugs in the most recent stable versions of all tested browsers. Some of the issues were unknown, other bugs were already reported or known. Below is a list of all the *new* bugs we found. This is harder to determine for browsers with closed bug trackers (Opera, IE) so in these cases we only list bugs not found by other major test suites.

We want to stress the fact that this table should not be used to compare web browsers. During development we used Mozilla SpiderMonkey's shell as the main JavaScript engine, so we have tested it more extensively than the other engines. The JavaScript shell also allowed us to disable certain test restrictions, so we were able to test it better than the browsers.

The third column lists the bug number, if available. The MB prefix means that the bug was reported to Bugzilla, Mozilla's bug tracker. Bugs reported to Opera have an OB prefix.

Affected browsers	Description	Notes
Firefox 4, Opera 11.50, Safari 5	parseInt should first convert its arguments to string. Some implementations don't do this when the argument is a number, this optimization is observable for very large or very small numbers (parseInt(3e100) should return the same value as parseInt("3e100")).	MB 653153, MB 663338, OB 344846
Firefox 4	A null character ("\0") is treated as the end of a property name.	MB 653175
Firefox 4	number.toExponential(undefined) should equal number.toExponential()	MB 653438
Firefox 4	Array.prototype.sort bug when the array has only undefined values and non-existant properties (it should move the undefined values to the front)	MB 653438

Firefox 4	Bug in the new JIT-compiler in Firefox 4: x === x should be false iff x is NaN, this failed when x was a local variable and the result of a negation operation	MB 646938
Firefox 4	On 64-bit builds the result of the expression -x === x was true when x is a positive zero but false when x is a negative zero (both should be true).	MB 652060
Opera 11.50	The expression JSON.stringify(2) === "2" returns false.	OB 340923
Opera 11. 50	Array.prototype.reverse bug with arrays with non-existant properties	OB 340927
Opera 11.50	Math.min.apply(null, x) returns a non-number value when x is an array containing a regular expression and a function object.	OB 340925
Opera 11.50	Object.prototype.toString.call(undefined) returns "[object Window]" instead of "[object Undefined]".	-
Opera 11.50	Opera crash (cause unknown).	OB 340761
Opera 11.50	Opera >>> operator bug in the JIT compiler.	OB 340935
Opera 11.50	Opera bug with || operator and constant folding.	OB 340943
Opera 11.50, Chrome 14	Array.prototype.toString does not call join.	-
IE 9	number.toExponential(undefined) throws a RangeError instead of including as many digits as necessary	-
IE 9	Math.max(x) and Math.min(x) return x, without converting x to number	-
IE 9	Object.prototype.valueOf does not apply ToObject (typeof Object.prototype.valueOf.call(true) is "boolean")	-

We also found many bugs in development versions of Mozilla's JavaScript engine. For instance, we reported more than 30 bugs found on the Type Inference branch, an experimental development branch which touches many parts of the engine. This demonstrates that our tool has also been very useful for quickly finding regressions.

Chapter 6

Conclusion and Future Work

We have created a new framework for testing JavaScript engines based on property-based testing. Although we have focussed on JavaScript, the underlying idea and design is language independent and can be used to test other languages. Despite the many existing test suites we have succeeeded in finding correctness bugs in every tested browser. Some of the issues we found were very recent regressions in builds not yet deployed to beta testers. This indicates that the test framework is especially useful for finding regressions quickly.

Another advantage is that our approach allows generating an endless amount of different tests. This process is cheap and can be fully automated, making it a good candidate for automatic regression testing after commits. In this regard our framework is very much comparable to fuzz testing.

There are still many ways to improve the test framework. It's important to have a good collection of *checks*, the properties used to generate correctness tests. Although we created an initial set of checks, mostly based on the ES5 specification, we were unable to write checks for everything in the specification due to time constraints. Furthermore, the test case generator could be extended to combine or interleave different checks to create more diverse tests. Instead of hardcoding most statement types in the generator, our custom expression generator approach could be extended to handle statements and make it easier to add support for new statements.

Bibliography

[1] Ecma International Technical Committee 39. ECMAScript Test262. http://test262.ecmascript.org/. [Online; accessed 5-July-2011].

[2] A.S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.

[3] Peter Bright. Lies, damned lies, and benchmarks: is IE9 cheating at SunSpider? http://arstechnica.com/microsoft/news/2010/11/lies-damned-lies-and-benchmarks-is-ie9-cheating-at-sunspider.ars. [Online; accessed 5-July-2011].

[4] I. Ciupa and A. Leitner. Automatic testing based on design by contract. In *Proceedings of Net. ObjectDays*, volume 2005, pages 545–557. Citeseer, 2005.

[5] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 35(9):268–279, 2000.

[6] Mozilla Corporation. Mozilla Security Bug Bounty Program. http://www.mozilla.org/security/bug-bounty.html. [Online; accessed 5-July-2011].

[7] Chris Leary (Mozilla developer). Mapping the monkeysphere. http://blog.cdleary.com/2011/06/mapping-the-monkeysphere/. [Online; accessed 5-July-2011].

[8] David Anderson (Mozilla developer). High Performance JavaScript: JITs in Fiefox. http://www.slideshare.net/iamdvander/intel-jit-talk. [Online; accessed 5-July-2011].

[9] Christian Plesner Hansen Erik Corry and Lasse Reichstein Holst Nielsen. Irregexp, Google Chrome's New Regexp Implementation. http://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html. [Online; accessed 5-July-2011].

[10] P. Godefroid, A. Kiezun, and M.Y. Levin. Grammar-based whitebox fuzzing. *ACM SIGPLAN Notices*, 43(6):206–215, 2008.

[11] Google. Chromium Vulnerability Rewards Program. `http://dev.chromium.org/Home/chromium-security/vulnerability-rewards-program`. [Online; accessed 5-July-2011].

[12] D. Herman. Parser API. `https://developer.mozilla.org/en/SpiderMonkey/Parser_API`. [Online; accessed 5-July-2011].

[13] Ecma International. Standard ECMA-262. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`. [Online; accessed 5-July-2011].

[14] B. Meyer. Applying design by contract'. *Computer*, 25(10):40–51, 1992.

[15] Jesse Ruderman. Introducing jsfunfuzz, August 2007. `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz`.

[16] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *ACM SIGPLAN Notices*, 44(2):37–48, 2009.

[17] Darrin Thompson. qc.js. `https://bitbucket.org/darrint/qc.js/`. [Online; accessed 5-July-2011].